BB

1

AFIT/GCE/ENG/93J-01

ANNS
An X Window Based Version of the AFIT Neural Network Simulator

THESIS

Ching-Seh Wu
Captain, ROCAF, Taiwan

AFIT/GCE/ENG/93J-01

Approved for public release; distribution unlimited

12pp

AFIT/GCE/ENG/93J-01

ANNS

An X Window Based Version

of the

AFIT Neural Network Simulator

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Ching-Seh Wu, B.S.

Captain, ROCAF, Taiwan

June, 1993

Preface

The objective of my research was to develop a neural network simulator using the techniques of modern software engineering. An X window based version of the AFIT Neural Network Simulator, or ANNS, is the result of this research. With respect to reach the goal of this thesis efforts, a hybrid software engineering paradigm which combines the best characteristics of the classic life cycle, prototype, and iterative methodologies was used for developments.

This thesis documents the analysis, design and implementation of the ANNS system. This system environment was developed for the study and research of dynamical changes in patterns of weight and nodes for artificial neural networks. Graphical representations of neural network algorithm simulations are displayed using X window based graphical routines. This system provides a user interface for neural network programmers that they can develop and add their own design of new neural network paradigms or algorithms into this system easily and become an integrated system.

I have many thanks to my thesis advisor, Dr. Steve K. Rogers, for his encouragement and enthusiasm, and allowing me almost complete freedom over the design and implementation of ANNS. I also wish to thank my committee members, Lt. Col. Phil Amburn and Dr. Denny Ruck, for their hints, suggestions, and editing of the draft manuscripts. I also want to thank Dr. Gregory L. Tarr for his wonderful *NeuralGraphics*.

I especially want to thank my beautiful wife, Pi-chiao (Joy) Yu, who basically lived almost two years as a single parent. I cannot repay her for the patience, understanding and support she provided during these twenty-seven months in United States. I also wish to thank my sixteen month old son, Kevin Wu, for making me smile when I needed it most. Finally, I want to extend my gratitude to everyone in

my family in Taiwan - my parents, my parents in law, my two brothers, and my two brothers in law and three sisters in law. Thank you very much !

Ching-Seh Wu

Table of Contents

List of Figures

Abstract

This thesis presents an X Window based neural network simulation environment developed at Air Force Institute of Technology (AFIT) using the techniques of modern software engineering . This artificial neural network simulator is a tool running on Sun SPARCstations and supporting two user modes: end-users and client-programmers. End-users interact with neural network paradigms developed by client-programmers for the purpose of studying and analyzing the execution of a particular Neural Network (NN) paradigm, or class of NN algorithms. Client programmers maintain the system and use this environment for the development of new NN paradigms or algorithms for end-users. The development follows a hybrid software engineering paradigm which combines the best characteristics of the classic life cycle, prototype, and iterative methodologies through *requirements, design, implementation, and testing.* An object-oriented approach is used for the design including preliminary and detailed design. The system is implemented with the C programming language on Sun workstation and uses the XView window-based environment. It provides users with a variety of control and input options: simulation speed control, multiple and simultaneous NN algorithm simulations, and simulation environment control.

ANNS

An X Window Based Version

of the

AFIT Neural Network Simulator

I. Introduction

An environment to examine internal operation of neural networks as they train could help determine the efficiency and accuracy of different topologies. Evaluation of internal constants and variables as the network trains may offer insight into which values may be best suited for a particular set of circumstances. In order to test and evaluate a number of paradigms and techniques, a neural network simulator with an X window based Graphical User Interface called the AFIT Neural Network Simulator (ANNS) was developed. The ANNS running on Sun SPARC workstations is a collection of software tools and demonstrations that provide graphical displays and allow users to interact with the network during execution of the training algorithms.

ANNS can provide a window so that the programer or user can view the dynamic behavior of an algorithm and its changes of learning state while the neural network paradigms or algorithms execute. Rather than simply viewing the static result of paradigm execution, ANNS presents paradigm execution as a series of tran sitions. As an educational tool, ANNS can help students understand different algorithms by providing a means for visualizing and interacting with algorithms as they execute. As a research tool, ANNS is useful in the development of new neural network paradigms, as well as the effective use of existing neural network algorithms.

1.1 Background

Many seemingly simple problems have proven intractable for ordinary computers using conventional algorithms. These problems seem simple because we solve them every day. Simple tasks like finding a light switch and turning it on would be trivial for a person but difficult for a computer. Other examples include navigating around a room, or selecting the best stocks and bonds to buy. Biological systems seem to have little trouble with these types of problems. For example, a common house fly has enough computational power in a few cells of protoplasm to fly straight at the ceiling, flip over and land upside down attaching itself to the surface with its little suction cup feet. Try that with a Cessna on autopilot.

Because biological systems seem so good at solving certain problems, many researchers have suggested building computers based on biological models. The results, for better or worse, have come to be grouped under the general heading of *artificial neural networks.* A real neural network is one of those things found in all animals for information processing.

Neural networks may offer a new approach to many problems which have proven intractable for many conventional algorithms. With the increased interest in finding neural network solutions to common problems, engineers and interested others, could benefit from a graphic software package to try out simple problems. That was the reason *NeuralGraphics* was built [51].

The *NeuralGraphics* system was developed to illustrate how to apply neural networks to a variety of problems. The system was initially implemented on Silicon Graphics IRIS 3130 system and has been moved to SGI IRIS 4D workstations. The environment includes demonstrations and applications of multi-layer feedforward networks using backpropagation, hybrid (joint supervised/unsupervised) training paradigms, radial basis functions, Hopfield associative memory, error surface analysis and network topology analysis. The program has been used on Silicon Graphics

system at the Air Force Institute of Technology and provided as a public domain tool for several years.

1.1.1 Artificial Perception . Artificial perception, as opposed to artificial intelligence, is the function of converting sensor measurement into symbols used by the intelligence system. Artificial perception allows a sensor to understand its environment.

Target identification and classification from electronic imagery and signal intelligence is a difficult problem due to the vast amounts of data involved. A single image can contain millions of bits of information, all of which need to be processed. Processing images for pattern recognition is a threefold problem. First, the targets must be separated from the background or segmented. Second, the data must be reduced to a manageable size, commonly called vector quantization or feature selection. This reduction in data can be accomplished by selecting specific features of a pattern and using only these features for classification. Good pattern recognition requires good features. Finally, the vectors must be classified. In most cases, the final classification is the easiest part of a pattern recognition problem.

Determining which features of an image form the best description of an object is a difficult problem. In addition to selecting the best features, the data must sometimes undergo significant preprocessing.

Success of a particular classification problem depends on a number of factors. First, consider the validity of the segmentation of the data. Has the actual target been separated from the background data and noise ? Is the feature extraction legitimate ? Do the features selected for the input vector represent a good description of the target ? Once the target has been extracted from the background, is the vector quantized description unique enough to allow classification ? Finally, is the neural network topology sufficient for the size of the decision region and can it accurately classify input pattern ? Special tools may be need to answer these questions.

1.2 Problem

The original intent for *NeuralGraphics* running on Silicon Graphics IRIS workstations was to provide a platform for neural network research. This never really transpired, for several reasons. First and foremost was a lack of available workstations. This problem has so far been addressed by using Sun Sparc stations, which are commonly used by the general AFIT engineering student body. Since these workstations run Openwindows (an X Window System-based graphical user interface), new students are indoctrinated into the Openwindows environment, and are therefore unfamiliar with Silicon Graphics system.

In the real world of AFIT neural network environment, there is no unified and integrated ANNS available for the end-users and client-programmers. End-users interact with neural network paradigms developed by client-programmers for the purpose of studying and analyzing the execution of a particular Neural Network (NN) paradigm, or class of NN algorithms. Client programmers maintain the ANNS system and use this environment for the development of new NN paradigms or algorithms for end-users. *NeuralGraphics* was originally implemented using the graphics library (GL) that comes with Silicon Graphics system and only supported a limited number of neural network paradigms. For future development of neural network paradigms, it will be difficult to integrate all new algorithms into a single simulator environment for client-programmers.

1.3 Research Objectives

This thesis effort resulted in the development of an X window based Graphical User Interface and integrated environment for controlling, displaying, and interacting with the neural network algorithms for ANNS on Sun Sparc stations. This dual purpose reflects the needs of two types of users: *"client-programmers" and "end-users"*.

Client-programmers are concerned with implementing the neural network paradigm or algorithm simulation with which end-users interact. With respect to client-programmers, the goal of this study is to create a program development which provides a consistent interface to the Neural network Simulator system and supports reusable software modules. The programmers should not have to reimplement modules common to several paradigms or algorithms, such as window management, user-interface, and display functions.

End-users view and interact with the simulations at a computer workstation. With respect to end-users, the goal of this investigation is to provide an neural network algorithm simulation run-time environment which provides a consistent method for interacting with the simulations. After simulating one neural network paradigm or algorithm, end-users should be able to simulate any neural network paradigm or algorithm, regardless of the type of algorithm or the client-programmer of the simulation.

In general, this investigation pursues the dual-interface approach. Develop a system through which a user can select, execute, and control individual simulations, each of which is a separate executable procedure. The client-programmer develops the executable procedures with the help of a library of neural network algorithm simulator support functions. The goal is to develop an X window based neural network simulation environment which presents an easy-to-use, functional interface to end-users, and provides an effective means for managing neural network simulations within the simulation environment for client-programmers.

1.4 Assumptions

The research and development efforts in this thesis were based on the following assumptions:

1. The code developed by Capt. Gregory L. Tarr correctly presents and displays the neural network paradigms and algorithms. [51]

2. The X window based Graphical User Interface is to be developed on Sun Sparc workstations using XView (X Window-System-based Visual/Integrated Environment for Workstations).
3. The C programming language is to be used for implementation.
4. This thesis effort does not include conducting any sensitivity analysis or validation and verification of any neural network paradigms or algorithms.
5. Since AFIT seems to have selected the Sun Sparc station platform as the engineering workstation standard, and since the user community will most likely remain AFIT for the foreseeable future, then portability is not a major issue at this time.

1.5 Approach

The basic approach to this thesis effort consists of the following steps:

1. Step 1 is a requirements analysis, including a review of current literature, such as conducting research in the areas of the Object-Oriented Design, Graphical User Interface, the X Window System and the *NeuralGraphics* status. Properly understanding these areas is essential in accomplishing the thesis effort. This is presented in Chapter II.
2. Step 2 is an analysis of *NeuralGraphics* to determine what steps can be taken to simplify the use of ANNS before the X window based Graphical User Interface (GUI) replacement phase begins. This is presented in Chapter III.
3. Step 3 is a development of a preliminary object-oriented design. The design includes the relationships among objects as well as their attributes and methods. This is also presented in Chapter III.
4. Step 4 is a development of detailed objected-oriented designs based on step 3. This is presented in Chapter IV.

5. Step 5 is an implementation stage, including a rewrite of the *NeuralGraphics* system Graphical User Interface based on Silicon Graphics GL library. Step 5 starts with a detailed discussion outlining the motivations for choosing XView and an analysis of currently available graphics libraries and their attractiveness to ANNS. These are presented in Chapter IV.

1.6 Summary

Autonomous military target detection and classification from electronic imagery is a topic of great importance to the Department of Defense of the United States. The solution to the problem may lie in one of several implementations of artificial neural networks. Several topologies for neural networks have been proposed, each of which provide a solution for a narrow class of pattern recognition problems. Some researchers (Huang and Lippmann) [11] feel that combinations of more than one type of neural network may result in a more dynamic and robust system. The goal of this thesis is to develop this kind of system called ANNS, an X window based Artificial neural network simulator.

The next chapter consists of a literature review of object-oriented design methods, Graphical User Interface, X window systems and *NeuralGraphics* status.

Chapter III describes the object oriented approach used for the requirements analysis and the initial design of the X windows based ANNS user environment.

Chapter IV provides the design and implementation of ANNS.

Chapter V presents the conclusions and recommendations.

In Appendix A, the *ANNS User's Manual* describes how to use the *AFIT Neural Network Simulator* system. Appendix B is the *ANNS Programer's Guide* ; it describes the ANNS system and presents a procedure for creating new paradigms and algorithms. Appendix C is *ANNS User Evaluation Form* which is a sample of the standard form used by the Department of Electrical and Computer Engineering

at AFIT to evaluate software systems. Appendix D includes the source codes of the ANNS system.

II. Literature Review

2.1 Introduction

The purpose of this chapter is to review some of the literature on object-oriented design, Graphical User Interface, X Window System, and the *NeuralGraphics* status. First, object-oriented design concepts are discussed, including the steps involved in conducting an object-oriented design as described by various authors. Advantages and strong points of object-oriented design are also discussed.

Secondly, the graphical user interface design concepts are also discussed. In order to begin developing a graphical user interface, we need to understand the user's requirements and the role the user interface is to serve in fulfilling those requirements. Once the "big picture" has been grasped, it is imperative that the user interface designer have some understanding of the factors involved in creating a satisfactory user interface. The last step in user interface development is to decide upon and master a computer language or system that can be used to effectively implement the design. Therefore, the basic concepts of the X Window System are described.

The X Window System is an industry-standard software system that allows programmers to develop portable graphical user interfaces. X allows programs to display windows containing text and graphics on any hardware that supports the X protocol without modifying, recompiling, or relinking the application.

Finally, the current status of the *NeuralGraphics* software package is discussed, including the basic structures and software design.

2.2 Object-Oriented Design

Object-oriented design (OOD) is based on a decomposition of the system into objects. This differs from functional decomposition techniques where the decomposition is based on functions. Each module in an object-oriented design is based on

an object whereas, in the functional decomposition, the modules are based on steps in the overall system process [2:211]. "Object-oriented design is a design method which is based on information hiding" [42:204]. Korson and McGregor state that "the object-oriented design paradigm takes a modeling point of view" [18:46].

2.2.1 Object-Oriented Design Concepts. Korson and McGregor describe five concepts in object-oriented methods. These concepts, which are described in the following section, are: "objects, classes, inheritance, polymorphism, and dynamic binding" [18:42].

2.2.1.1 Object. Booch defines an object as "something you can do things to. An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class. The terms instance and object are interchangeable" [5:516]. The behavior of an object is "characterized by the actions that it suffers and that it requires of other objects" [2:211]. "The intent of an object is to represent a problem domain entity" [42:4-57]. For example, the coordinate of a point on a workstation screen is *(20, 20)* . This point can be defined as an single object with *display* operation on it. Since there are many points with different coordinates on the screen and each of them is a individual object, the *point* can be defined as an object class as shown in figure 1.

Objects use memory and have an associated address. Associated with an object are procedures and functions which define the operations on the objects. [18:42] "Objects communicate by passing messages to each other and these messages initiate object operations" [42:204]. Communication may be asynchronous. OOD is an excellent method to use in designing parallel or sequential programs. [42:204]

2.2.1.2 Class. A class is "a set of objects that share a common structure and a common behavior. The terms class and type are usually (but not always) interchangeable; a class is a slightly different concept than a type, in that it

emphasizes the importance of hierarchies of classes" [5:513]. "From the point of view of a strongly typed language, a class is a construct for implementing a user-defined type" [18:42]. For example, *Line, Arc, Polygon, and Circle* are object classes in figure 1.

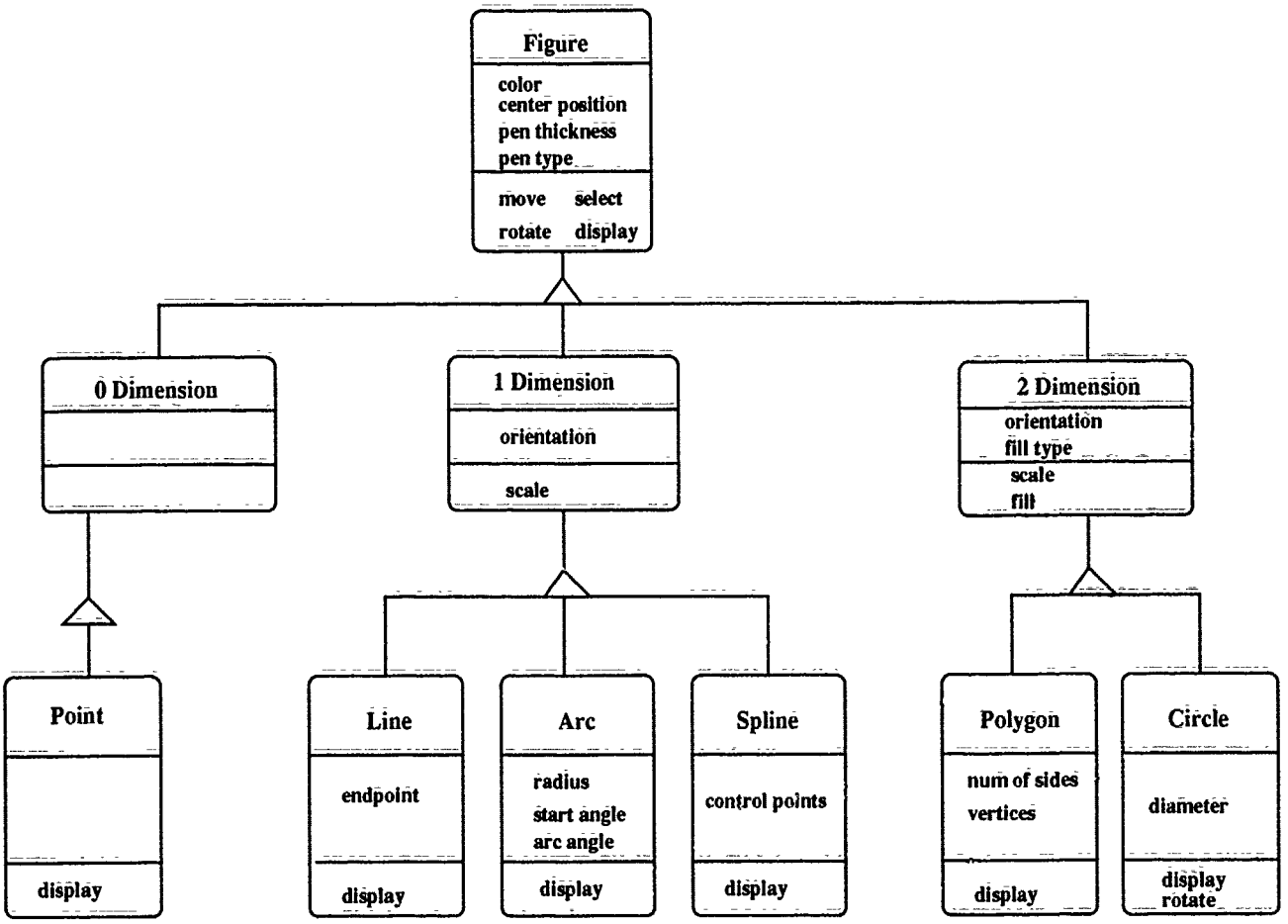


Figure 1. Inheritance for graphic figures

Object-oriented techniques use an Abstract Data Type (ADT) to represent a class of objects. According to Booch, an ADT "denotes a class of objects whose behavior is defined by a set of values and a set of operations, including constructors, selectors, and iterators" [3:216]. "Ideally, a class is an implementation of an ADT. This means that the implementation details of the class are private to the class" [18:42].

2.2.1.3 Inheritance . "Inheritance is a relation between classes that allows for the definition and implementation of one class to be based on that of other existing classes" [18:43]. "Inheritance defines a 'kind of' hierarchy among classes in which a subclass inherits from one or more superclasses; a subclass typically augments or redefines the existing structure and behavior of its superclasses" [2:154]. Korson and McGregor state that the inheritance relation often denotes an "is a " relation. Inheritance supports reuse of software components. [18:43-44] Figure 1 shows classes of graphic geometric figures. *Figure* object class has *0 Dimensional, 1 Dimensional,* and *2 Dimensional* figures. *Move, select, rotate,* and *display* are operations inherited by all subclasses. *Scale* applies to one- and two-dimensional figures. *Fill* applies only to two-dimensional figures.

2.2.1.4 Polymorphism. Polymorphism is defined as "a concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways" [5:517]. In other words, this means that polymorphism is a technique in which an object can have more than one form. "A polymorphic reference has both a dynamic and a static type associated with it. The 'is a' nature of inheritance is tightly coupled with the idea of polymorphism in a strongly typed object-oriented language" [18:45]. The same operation may apply to many different classes. Such an operation is *polymorphic;* that is, the same operation takes on different forms in different classes. A *method* is the implementation of an operation for a class. For example, the class *File* may have an operation *print.* Different methods could be implemented to print ASCII files, print binary files, and print digitized picture files. All these methods logically perform the same task-printing a file.

2.2.1.5 Dynamic Binding. Booch defines dynamic binding as "a binding in which the name/class association is not made until the object designated by the name is created (at execution time)" [5:513]. Binding, as defined by Booch, "denotes the association of a name (such as a variable declaration) with a class" [5:513]. Korson and McGregor state that dynamic binding "means the code associated with a given procedure call is not known until the moment of the call at runtime" [18:46]. Dynamic binding "is associated with inheritance and polymorphism in that a procedure call associated with a polymorphic reference may depend on the dynamic type of that reference" [18:46].

2.2.2 A Sample Object Model. This section provides an object model (associated with ANNS system design) of a workstation window management system, such as the X Window System, as an example to illustrate the concepts of object-oriented design. Figure 2 describes many object modeling constructs and shows how they fit together into a large model.

Class *Window* defines common parameters of all kinds of windows, including a rectangular boundary defined by the attributes *x1, y1, x2, y2,* and operations to display and undisplay a window and to raise it to the top (foreground) or lower it to the bottom (background) of the entire set of windows. *Panel, Canvas,* and *Text window* are varieties of windows. A canvas is a region for drawing graphics. It inherits the window boundary from *Window* and adds the dimensions of the underlying canvas region defined by attributes *cx1, cy1, cx2, cy2.* A canvas contains a set of elements, shown by the association to class *Shape.* All shapes have color and line width. Shapes can be lines, ellipses, or polygons, each with their own parameters. A polygon consists of an ordered list of vertices, shown as an aggregation of many points. Ellipses and polygons are both closed shapes, which have a fill color and a fill pattern. Lines are one-dimensional and cannot be filled. Canvas windows have operations to add elements and to delete elements.

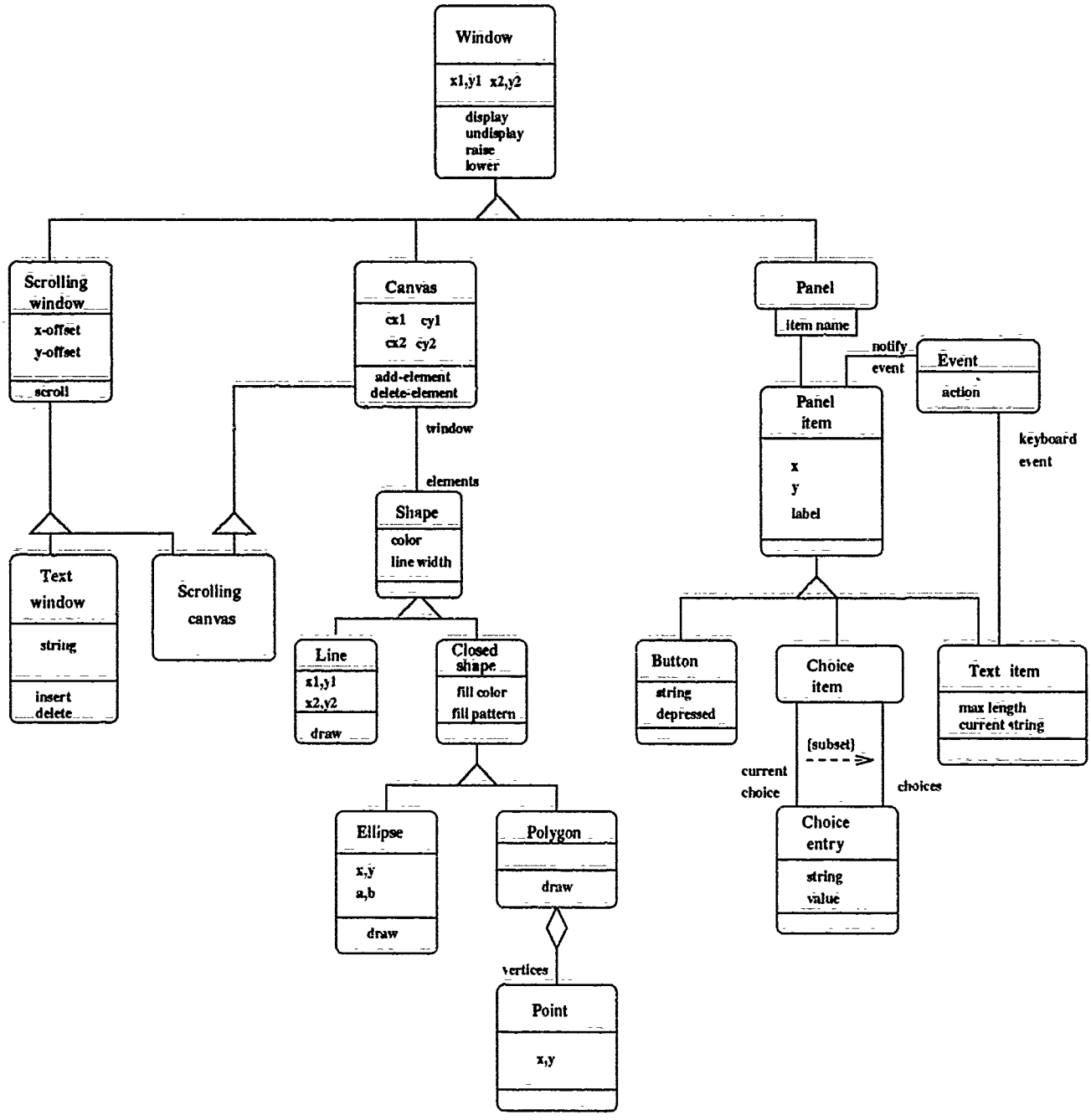


Figure 2. Object Model of Windowing System

Text window is a kind of a *Scrolling window,* which has a 2-dimensional scrolling offset within its window, as specified by *x-offset* and *y-offset,* as well as an operation *scroll* to change the scroll value. A text window contains a string, and has operations to insert and delete characters. *Scrolling canvas* is a special kind of canvas that supports scrolling; it is both a *Canvas* and a *Scrolling window.* This is an example of *multiple inheritance.*

A *Panel* contains a set of *Panel item* objects, each identified by a unique *item name* within a given panel, as shown by the qualified association. Each panel item belongs to a single panel. A panel item is a predefined icon with which a user can interact on the screen. Panel items come in three kinds: buttons, choice items, and text items. A button has a string which appears on the screen; a button can be pushed by the user and has an attribute *depressed.* A choice item allows the user to select one of a set of predefined *choices,* each of which is *Choice entry* containing a string to be displayed and a value to be returned if the entry is selected.

When a panel item is selected by the user, it generates an *Event,* which is a signal that something has happened together with an action to be performed. All kinds of panel items have *notify event* associations. Each panel item has a single event, but one event can be shared among many panel items. Text items have a second kind of event, which is generated when a keyboard character is typed while the text item is selected. Association *keyboard event* shows these events. Text items also inherit the *notify event* from superclass *Panel item;* the *notify event* is generated when the entire text item is selected with a mouse.

2.2.3 Object-Oriented Design Process. Different authors describe different steps to use in conducting an object-oriented design. What one author calls an object-oriented design, another author calls object-oriented development or requirements analysis. Since many authors use a modified version of Booch's object-oriented design process, his steps will be discussed in a later section of this chapter. This

section discusses a method described by Henderson-Sellers and Edwards which they call an object-oriented development methodology.

Henderson-Sellers and Edwards describe seven steps used by Bailin in his object-oriented requirements specification method. They state that these steps could "obviously transcend the requirements stage well into detailed design" [10:148]. Bailin's seven steps, as described by Henderson-Sellers and Edwards, are [10:148-149]:

1. identification of key problem space objects,
2. distinguish between active and passive objects,
3. establish data flows between active objects,
4. decomposition of objects into "sub-objects",
5. check for new objects,
6. group functions under new objects,
7. assign new objects to appropriate domains.

According to Henderson-Sellers and Edwards, Bailin sees the first three steps as ones which are accomplished only once, while the other steps are performed iteratively. Henderson-Sellers and Edwards propose a "seven-point methodological framework for object-oriented systems development" [10:149]. The steps, and a description of each follow [10:140-150]:

1. Undertake object-oriented systems requirements specification. "This stage is high-level analysis of the system in terms of objects and their services, as opposed to the system functions" [10:149].
2. Identify the objects and the services each can provide. This equates to the entities and their interfaces. "This is where the functional features will be defined; although no indication of implementation is required" [10:150]. Henderson-Sellers and Edwards propose that an object dictionary be established. The

visible interface is defined by identifying the objects, and the operations on the objects, as well as the services offered.

3. Establish interactions between objects in terms of services required and services rendered. Henderson-Sellers and Edwards suggest that an entity-data flow diagram (EDFD) or entity-relationship diagram (ERD) be used for this step. They suggest that a better name for this diagram is an information flow diagram (IFD).

4. Use of lower-level IFDs. This is where analysis and design merge. The lower-level IFDs show "more internal details of the objects" [10:150]. From this step on, bottom-up concerns should be analyzed.

5. Bottom-up concerns. During this step, objects are constructed from libraries of previously used objects. Implementation of low-level classes begins.

6. Introduce hierarchical inheritance relationships as required. This step involves determining whether there are any superclasses or new subclasses. Henderson-Sellers and Edwards propose the use of an inheritance diagram to show the inheritance relationships. They state that this step is needed to provide a well-defined hierarchy so that future efforts can reuse the resulting structure.

7. Aggregation and/or generalization of classes. This step might require reviewing and modifying the IFDs. Prototyping might begin at this stage. The identified system classes can undergo another stage of development which Henderson-Sellers and Edwards call generalization. "At this stage the components continue to be worked on until they are general, generic, and robust enough to be placed in a library of components" [10:150].

2.2.4 Booch's Object-Oriented Design Process. This section describes the five steps of Booch's design process as describe in his book, *Software Components*

with Ada [1][3]. Since other authors use very similar steps, it includes information from various authors.

2.2.4.1 Identify the Objects and Their Attributes. This step involves taking a narrative requirements document and extracting the nouns, pronouns, and noun phrases [3, 6, 13]. Some objects may be similar to the other objects.

In this case, a class of objects is formed [4:48]. Once all the objects and classes are identified, a decision must be made as to whether they will be kept or discarded [13:44]. Just because an object is identified from the requirements document does not mean that it should become part of the design and implementation [13:44].

Once the list of objects is refined, then the attributes of the objects should be determined. "The attributes of an objected characterize its time and space behavior" [3:17]. Jean and Strohmeier state that "these properties are given by the qualifiers of the objects and classes within the informal strategy and by the additional information found in the requirements analysis document" [13:44]. EVB Software Engineering, Inc. states that these are the "adjectives and adjectival phrases" [3:2-8].

2.2.4.2 Identify the Operations Suffered By and Required of Each Object. In this step, the requirements document is used to extract verbs, verb phrases, and predicates [6, 13]. Then, the extracted verbs, verb phrases, and predicates are associated with a particular object [6, 13]. Jean and Strohmeier say "The goal is to bind each operation to a single object or a single class" and that "no operation should be left alone" [13:45].

"The operations suffered by an object define the object's activity when acted upon by other objects". By defining the operations required by an object, an attempt is made to decouple objects from one another. [3:17]

[1]Since Xlib and Xt of X window system are written in C language. We need to translate the OOD concepts of Ada implementation to the C language implementation.

During this step, a determination should be made as to whether the operation is a selector, a constructor, or an iterator [13:45]. A selector evaluates the current object state; a constructor alters the state of an object; an iterator permits all parts of an object to be visited [13:20].

2.2.4.3 Establish the Visibility of Each Object in Relation to Other Objects. As part of this step, a decision is made as to what objects "see" and are "seen" by other objects [4:49]. The dependencies among objects should be established [2:219]. This can be done diagrammatically by drawing each object and then connecting the objects with a line to show the visibility between the objects [3:28].

EVB divides Booch's step into four substeps. The first substep is to decide on how to implement the operations. Subprograms, packages, tasks, and generics are the program units used to implement an object. The second substep formally describes the interfaces among the objects. These descriptions can be textual or graphical. A program unit which depends on another program unit must be compiled after the first program. This substep helps determine the compilation order. EVB's third substep is to create any additional objects and operations which are needed to help the implementation strategy. These items are ones that were not identified as part of the informal strategy but must be visible outside of the program unit. The last substep is to produce graphical annotations to represent the formal strategy. The diagrams give no indication as to how an object should be implemented nor do they show much about the underlying implementation of the operations. The diagrams serve as a map for the software engineer to follow throughout the rest of the design process. [6].

2.2.4.4 Establish the Interface of Each Object. This step is accomplished by writing a module specification for each object. Booch states that "this specification also serves as a contract between the clients of an object and the object itself". [3:18]

2.2.4.5 Implement Each Object. This "involves choosing a suitable representation for each object or class of objects and implementing the interface from the previous step" [3:18]. An object is implemented in C "as a structure set of procedures and internal data" [35:342].

2.2.5 Advantages and Strong Points of Object-Oriented Design. Sommerville describes the following advantages to OOD [42:205]:

- Message passing eliminates the need for shared data areas for communication between objects. Overall system coupling is thus reduced.
- All state and representation information is kept within the object itself, making the object an independent entity that may be readily changed. Objects can not access information on other objects either deliberately or accidentally. Changes may be made without reference to other system objects.
- Objects may execute either in parallel or sequentially. They may also be distributed. The decision as to whether parallelism should be used does not need to be made at an early stage of the design process.

Korson and McGregor describe seven ways in which object-oriented design provides support for a good design.

1. *Modularity.* Classes become the modules. "This means that not only does the design process support modularity, but the implementation process supports it as well through the class definition". [18:50]
2. *Information Hiding.* "The class construct supports information hiding through the separation of the class interface and the class implementation" [18:51]. This separation permits the class specification to be mapped to various implementation and means some maintenance can be accomplished without the user's knowledge [18:51].

3. *Weak Coupling.* Object-oriented design supports weak coupling [18:51]. Since classes are designed as a collection of objects and the operations on those objects, the "interface operators of a class are inward-looking in the sense that they are intended to access or modify the internal data of the class" [18:51]. This leads to less coupling which is desirable.

4. *Strong Cohesion.* Strong cohesion is desirable and supported by object-oriented design. Korson and McGregor state that "a class is a naturally cohesive module because it is a model of some entity" [18:51]. Functional cohesion is desirable form of cohesion. Booch defines it as cohesion "in which the elements of a class or module all work together to provide some well-bounded behavior" [5:124]. OOD supports functional cohesion. The fact that OOD makes use of inheritance does not mean that the cohesion is weakened since both the data and functions which are inherited from another class form a natural group [18:51]. These natural groups are "brought together to represent one concept" [18:51].

5. *Abstraction.* Object-oriented design supports abstraction. Booch defines abstraction as "the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply-defined conceptual boundaries relative to the perspective of the viewer" [5:512]. Korson and McGregor discuss two types of abstraction which support OOD: abstraction by specification and abstraction by parameterization [18:51-52]. Abstraction by specification separates the specification of an object from its implementation [18:52]. "Abstraction by parameterization abstracts the type of data to be manipulated from the specification of how it is to be manipulated" [18:52]. Seidewitz and Stark state that there is a "spectrum of abstraction" including entity, action, virtual machine, and coincidental abstraction, which in conjunction with information hiding, provide the main guidance for defining objects [38:4-57]. Entity abstraction, which is the best level, is where an object "represents a use-

ful model of a problem domain entity" [38:4-57]. Action abstraction is where "an object provides a generalized set of operations which all perform the same kind of function" [38:4-57]. Seidewitz and Stark describe virtual machine abstraction as the case in which "an object groups together operations which are all used by some superior level of control or all use some junior level set of operations" [38:4-57]. The worse level of abstraction is the coincidental. This level of abstraction is defined as where "an object packages a set of operations which have no relation to each other" [38:4-57].

6. *Extensibility* Object-oriented methods are "easily extended" [18:52]. Inheritance supports this in two ways. First, because inheritance permits "the reuse of existing definitions to ease the development of new definition" [18:52]. Second, the polymorphic property also supports extensibility in designs [18:52].
7. *Integrable.* Designs produced by OOD "facilitate the integration of individual pieces into complete designs" [18:52]. This includes both the use of classes and objects [18:52].

Booch discusses coupling, cohesion, sufficiency, completeness, and primitiveness as means of determining that a design is good. Coupling and cohesion were discussed above. By sufficiency, Booch "means that the class or module captures enough characteristics of the abstraction to permit meaningful and efficient interaction" [5:124]. Completeness means "that the interface of the class or module captures all of the meaningful characteristics of the abstraction" [5:124-125]. Completeness is a subjective matter and should not be overdone. Primitiveness implies that an operation can be implemented if the developer is given access to the underlying representation of the ADT. [5:124-125]

2.2.6 Object-Oriented Design and Simulation . "The object-oriented design of simulations is based on the concept of abstract data types" [55:123]. Object-oriented techniques lend themselves to simulation because the "things" which should

be modelled are objects and what each of the "things" can do are the operations on the objects [31:278]. This defines an abstract data type. Roberts and Heim state that an "object-oriented design attempts to bridge the gap between the model and what is modeled" [31:278]. They also state that "division into classes, recognition of methods, and the organizations of hierarchies from the basic approach to object-oriented modeling" [31:279]. Methods are the operations performed on an object.

One benefit of an object-oriented simulation system is the focus on objects. Focusing on objects provides both data abstraction and information hiding which help to modularize the system. This "stimulates the user to identify the principal components of a system and to specify their behaviors and interactions". [31:279]

Another benefit of an object-oriented simulation is that existing models can form the basis for new models. By using overloading and inheritance, old objects can take on new meanings. [31:280]

The resulting amount of code generated using object-oriented simulations is less than using traditional approaches. This makes it easer to manage the model and also permits models to be larger and more realistic. [31:280]

Objects provide a natural starting point for concurrency [31:280]. Concurrency permits more than one object to be processing at the same time as long as the objects do not need to communicate with each other.

2.3 Graphical User Interfaces

The user interface is the component of the application through which the user's actions are translated into one or more requests for services of the applications, and that provides feedback concerning the outcome of the requested action [25]. The design of efficient and easy to use interfaces is receiving increased attention these days. Most people now realize that if an application has a user interface that is "unfriendly" or difficult to use, it is probably going to sit on the shelf unused.

2.3.1 User Interface Design. While much has been written recently on the subject of user interface design, it is hard to define exactly what is meant by a "good" user interface. Often, the closest one can come to a definition is an enumeration of qualities a user interface should have. Accordingly, it is not easy to design a user interface. Brad Myers describes user interface design as more of an art than a science. However, he does list some things to consider when producing a design [24]:

- *Learn the application.* In order to determine what data to display and how best to display it, the designer must have a good understanding of the functionality of the system. This is often one of the most significant steps in interface design as a poor understanding can be difficult to overcome once the design progresses.
- *Learn the user.* The designer must determine the skill levels of the intended users, their backgrounds, and the amount of training likely to be needed.
- *Learn the hardware and environmental constraints.* Is the system going to be run on a particular type of machine ? Will special input or output devices, such as mice, terminals, or plotters, be used ?
- *Evaluate similar products.* The designer should study the user interface of similar systems and of systems in the same environment.
- *Determine the support tools.* There are many toolkits available to assist in the design and implementation of user interfaces. Also, user interface management systems (UIMS) are becoming more popular as a means of increasing productivity in the user interface design.
- *Plan to incorporate Reset, Quit, and Help from the beginning.* It is very difficult to try to add these functions after the system is under development. The nature of the actions impacts the design of the application's data structures.

- *Separate the user interface from the application.* The user interface and the application should be modularized with design of the former being based on the functionality of the latter.
- *Design for change.* The user interface will change more than the functionality of the application. These changes frequently will be based on customer reaction to the delivered system.

Two of the items in the above list deserve a broader discussion. These are the support tools and the separation of the application from the interface. As previously mentioned, the two major types of tool for user interface design are toolkits and user interface management systems (UIMS). One problem with toolkits is that it is often difficult to determine what part of the toolkit to use to perform a particular function. Furthermore, since the work must be done over and over with each new application, consistency between systems is in jeopardy [17]. UIMS, on the other hand, are designed to aid in "rapid development, tailoring and management of the interaction in an application domain across varying devices, interaction techniques and user interface styles" [20:33]. This may include such things as handling user errors, providing helps and prompts, and validating users inputs.

Separating the user interface software from the application software has many attractive benefits. Typical user interface design consists of one or more prototypes offered to the user for review. The user then evaluates the interface and offers suggestions for improvement. If the application and user are closely interwoven the user interface designer may have difficult making the suggested improvements. The job can be much easier, however, if the functionality of the application is separated from the user interface. Pedro Szekely lists the following benefits of minimizing dependencies between the application and the interface [50:45]:

- The user interface can be packaged into components that can be reused in other interfaces.

- The user interface can be changed without impacting the functionality.
- Multiple user interface can be developed for a single application, each one tailored to a different class of users, or to a different set of input and output devices.
- The functionality of an application can be called from another program directly, without simulating the input required by the user interface.
- The user interface can be specified by means other than programming, for example, by interactively drawing and demonstrating how the interface should behave.

2.3.2 User Interface Characteristics. Whatever design method is used, effective user interfaces frequently have certain qualities. Brad Myers lists the following attributes of so-called "good" user interface [23]:

- *Invisibility:* The user interface should be transparent to the user, such that the user has the sense that he is directly manipulating "real" objects on the screen. The user interface should not interface with the operator's concentration on the task being performed.
- *Minimal training requirements:* No more than one hour of training should be necessary before the user can be productive on the system.
- *High transfer of training:* The system's appearance and performance should be similar to other systems dealing with the same subject matter. This external consistency between systems will help reduce training times when switching from system to system.
- *Predictability:* The objects and operations should perform similarly across contexts of the system. This internal consistency leads to a system where user can anticipate how the computer will behave.

- *It is flexible:* The user interface should allow user to operate in the manner with which they are most comfortable. Users should be able to customize certain attributes to their own style and taste.

2.3.3 User Guidance . User guidance refers to system documentation, the on-line help system, and messages sent as a result of user actions. This is an area that does not always receive the attention it deserves. However, it should be considered at every stage of interface design because of the significant contributions it can make to effective system operation [41]. According to Smith and Mosier,

> The fundamental objectives of user guidance are to promote efficient system use (i.e., quick and accurate use of full capabilities), with minimal time required to learn system use, and with flexibility for supporting users of different skill levels. [41:291]

Often, the first impression a user gets of a system is from error messages [42]. Thus, the interface designer should make an effort to write error messages that are both polite and constructive without being offensive. When possible, the error message should suggest how the user might recover from the error. Also, the user should have the option of getting a help message to give insight as to the cause of the error.

It is difficult for an interface designer to anticipate the level of help users will need. To accommodate all types of users, the help system should provide different levels of help. When the user first requests help, the system should provide a brief overview of the topic and give the user capability to request a continuation of the help. Each successive level of help would give greater detail on the subject [54].

2.4 The X Window System

User interfaces using some type of windowing system are fast becoming a common feature of most computer systems. As a result, users tend to expect all application programs to have a professional, polished user-friendly interface. [59] The X

Window System provides the mechanism to achieve this goal as well as many others described in the previous section.

The X Window System, or X, is a device independent, network transparent windowing system that allows for the development of portable Graphical User Interfaces (GUIs) [28, 37, 27]. It was developed in the mid 1980's at the Massachusetts Institute of Technology (MIT) in response to a need to execute graphical software on several different types of different workstations. Robert Scheifler of MIT and James Gettys of Digital Equipment Corporation (DEC) developed X with the primary goals of portability and extensibility [37]. Another major consideration was to restrict the applications developer as little as possible. As a result, X "...provides mechanism rather than policy"[14:xvii].

To achieve the goals, the X Window System relies on the fundamental principles of network transparency and a request/event system. Software toolkits are then layered on top of the basic system to provide an easier programming environment.

2.4.1 X Window System Principles .

2.4.1.1 Network Transparency . Oliver Jones describes network transparency as the capability for X application programs running on one CPU to show their output and receive their input "...using a display connected to either the same cpu, or some other cpu"[15:4]. The X Window System achieves this transparency using a client-server model. In X, each workstation that is to display graphical information(i.e.,windows or their contents) must have a process called the X server. According to Douglas Young, the X server "...creates and manipulates windows on the screen, produces test and graphics, and handles input devices such as a keyboard and mouse" [59:2].

The core of the X system is the server. The server allocates and manages all the necessary data structures required to support a screen. There is one server

per cpu, but a server can manage more than one screen (analogous to a file server with diskless clients). Applications programs using the server are known as clients. Any application which complies with the X protocol (an asynchronous byte-stream protocol) can communicate with the server. Obviously, a server can connect to many clients, but a client can also connect to more than one server. A client and server need not be on the same machine, or even the same network.

The server provides the device independent interface to the platform on which it resides. A specific version of the server must be installed for each platform. For example, in a networked workstation environment, each workstation has a device dependent server running in the background controlling the screen.

Clients and servers use the X protocol to communicate with each other over a network. As figure 3 shows, many clients can connect to a single server. Although not shown, a client can also be simultaneously connected to several X servers. In X, the client(s) and server can reside on the same physical machine, or they may be on the separate machines.

2.4.1.2 Requests and Events. The network protocol mentioned in the last section is the method with which clients and servers communicate. This section discusses the mechanisms used to carry out the communication. The clients and servers communicate with each other by sending requests and event notifications, respectively.

When a client wants to perform some action on the display, it communicates this desire by issuing a request to the appropriate X server. Young states:

> Clients typically request the server to create, destroy, or reconfigure windows, or to display text or graphics in a window. Clients can also request information about the current states of windows or other resources. [59:4]

The X server, conversely, communicates with the clients by issuing event notifications. Event notifications are sent in response to such user action as moving a

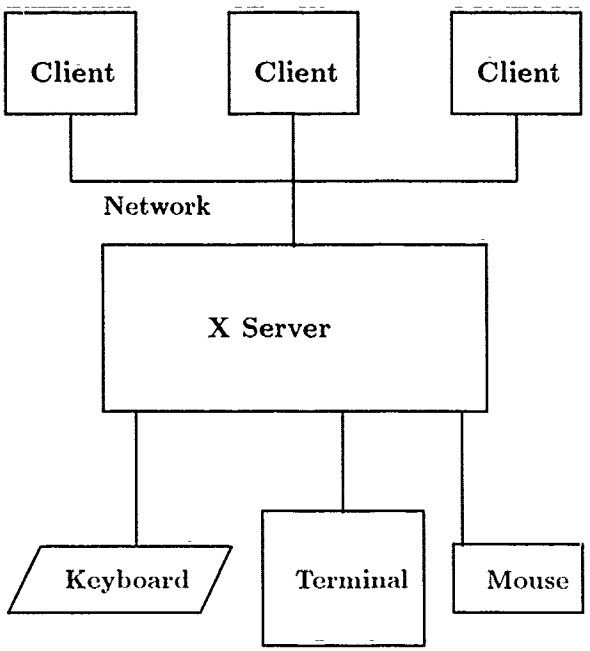


Figure 3. The X Client-Server Model

mouse into a window, by pressing a mouse button, or pressing a key on the keyboard. The X server also sends event notifications when the state of a window changes [27]. Applications programs act on these events by registering callbacks with the X Window System. A callback is simply a procedure or function that is to be executed when a specific event occurs.

Because of the reliability of the network, events and requests are sent asynchronously and data can be sent in both directions simultaneously [36]. This configuration makes for faster communication since the clients can send requests at any time and need not wait for an acknowledgement. The protocol guarantees the messages will be received in the proper order. Furthermore, there is no need for clients to continuously poll the server for information. "Instead, clients use requests to register interest in various events, and the server sends event notifications asynchronously" [36:xviii].

2.4.1.3 Basic Components. The X Window System was designed to provide the mechanism for the application program to control what is seen on the display screen. The programmer is not constrained by any particular policy. These mechanisms are embodied in a library of C functions known as Xlib. The Xlib routines allow for client control over the display, windows, and input devices. Additionally, the functions provide the capability for clients to design such things as menu, scroll bars, and dialogue boxes. Most X application programs make use of a special client program called window manager. The program utilizes the mechanisms of Xlib to relieve the application program of such tasks as moving or resizing windows [36]. Brad Myers writes that a window manager helps the user monitor and control different activities by physically separating them into windows on the computer screen [22].

Figure 4 represents the most basic X environment. In this diagram, an application program and a window manager operate as separate clients connected to a single server.

2.4.2 Toolkits. While application programmers can use the Xlib routines to accomplish any task in X, many find the low-level routines tedious and difficult to use. To simplify the development of applications programs, many toolkits have been developed. Toolkits can be viewed as libraries of graphical programs layered on top of Xlib. They were designed to hide the details of Xlib, making it easier to develop X applications. [26]

There are several toolkits available today. Some of the better known ones include: the X Toolkit (Xt) from MIT, the Xlib Toolkit (Xr) from Hewlett-Packard (HP), Open Look and XView from Sun Microsystems, and Andrew from Carnegie Mellon University. Of those listed, Xt is one of the most popular [30]. Along with Xlib, it is delivered as a standard part of the X window System.

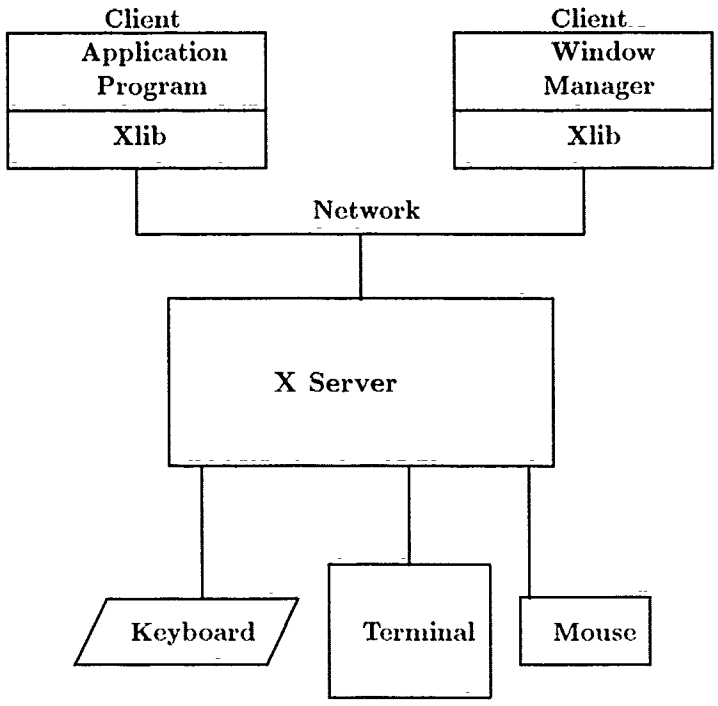


Figure 4. Basic X Environment

Xt is an objected-oriented toolkit used to build the higher level components that make up the user interface [30]. It consists of a layer called the Xt Intrinsics along with a collection of user interface components called widgets. Widget sets typically consist of objects such as scroll bars, title bars, menus, dialogue boxes and buttons. In keeping with the X philosophy, the Xt Intrinsics layer remains policy free. As such, it only provides mechanisms that do not affect the "look and feel" (outward appearance and behavior) of the user interface [59]. These mechanisms allow for the creation and management of reusable widgets. It is this extensibility along with its object-oriented design that makes the X Toolkit attractive to user interface designers.

It is the programmer's choice of a widget set that determines the high-level "look and feel" of the user interface. Just as there is no "standard" toolkit, there are many different widget sets supported by Xt Intrinsics. However, as Young writes,

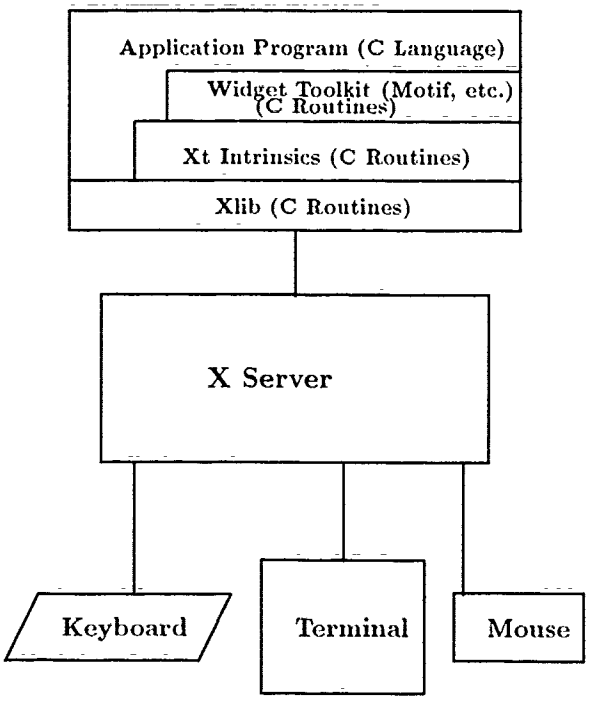


Figure 5. Typical X Windows Configuration

"...from an application programmer's viewpoint, most widget sets provide similar capabilities" [27:12]. Some of the more popular widget sets include the Athena widget set from MIT, the X widget set from HP, and the Motif widgets from the Open Software Foundation.

Structurally, the Xt Intrinsics is built on top of Xlib. The XView widget set, in turn, relies on the functions provided by the Xt Intrinsics. A typical application program may make calls to the widget set, the Xt Intrinsics, or Xlib itself during its execution. This configuration is illustrated in figure 5.

Many user interface designers elect to design their own widget sets. Some do it for the challenge. Others design their own widgets out of necessity. A user interface designer may have a need for a special widget not provided by any available widget sets. However, designing custom widgets decreases the portability of the user interface code and of the application code in general [9].

2.5 *The current status of the NeuralGraphics*

The *NeuralGraphics* system was designed for the Silicon Graphics IRIS workstation and written in the 'C' programming language. The environment includes demonstrations and applications of Kohonen mappings, multi-layer feedforward networks using backpropagation, hybrid (joint supervised/unsupervised) training paradigms, radial basis functions, hopfield associative memory, error surface analysis and network topology analysis. The *NeuralGraphics* software package consists of independent programs run from a common menu. The programs are selected from a shell or script program which calls the individual programs. Each program is independent and is run as an execution file from the script program. This section will discuss the modules which make up the *NeuralGraphics* package. [51]

2.5.1 Basic Structures and Design Considerations. Each of the neural networks of the *NeuralGraphics* package is composed of a number of independent software modules. In addition to the basic modules, a graphics tool box is provided, which is specific to the Silicon Graphics IRIS system.

At the highest level of abstraction, a neural network simulation consists of two loops: the network loop which is controlled by the programmer, and the event loop which is usually associated with the hardware. The event loop is the hardware and software structure which watches the input devices, i.e. keyboard and mouse for activity.

The neural network loop consists of: a routine to select the input vectors, a propagation algorithm to feed the vector through the network and compute the output of all the network nodes, a training routine for modification of weights, a graphic-display function and an analysis routine for periodic testing.

The event loop is concerned with the hardware and monitors the keyboard for the keys being pressed or the mouse being moved. This loop allows the user to

change the flow of control in the training process to for such functions as saving the weights, changing the network topology, or eliminating nodes.

Outside both loops is an important lower level modules which includes the initialization procedure for the graphics hardware and an event driven menu to control the training and operation of the network.

In pseudo-code the general flow of the network is shown below [51]:

```
begin
INITIALIZE
loop {
     MAKE_INPUT
     PROPAGATE
     TEST
     TRAIN_NET
     DISPLAY_NET
     Event Handler}
end loop
end
```

2.5.2 Objects and Operations. The OOD design of *NeuralGraphics* package was done by creating two types of software elements: objects and operations. The NET object is made up of a number subobjects, some obvious, like layers, nodes, and weights, and some not so obvious, like pointers to propagation and training rules, and connection information [51]. In parallel to the objects are the operations, or those procedures which act upon the objects. In this case, the net can be *displayed* or a layer can be *propagated, updated, tested* etc.

The procedure uses an object in the form of a data structure: a network and an array of layers.

```
typedef struct{
              int layers;
              float *input;
              float *output;
              float *desired_output;
              short *out_mask;
```

```
 layer *layer[10];
} NET;
```

NET contains variables to declare its input, output, desired output and an array of layers. Layers are a substructure to NET [51].

```
typedef struct{
                int size_input, size_output;
                float *input;
                float **weights;
                float *mask, *out_mask;
                float *output;
                float *theta;
                float *delta;
                float **momentum;
                void (*update)();
                float (*propagate)();
                } LAYER;
```

The NET object was made up almost entirely of pointers. The only element that is actually stored in the NET is the actual number of layers.

2.5.3 Initialization Modules. The initialization routine has two functions. The most important is the establishment of the network in memory. In addition, the weights, and thresholds are filled either by a random number generator or by a stored file from a previously trained net. The second function is the equipment check to determine the nature of the graphic displays and to initialize the video drivers as necessary. The size of the screen, color planes, and graphic capabilities would determine exactly how much data can actually be displayed. [51]

2.5.4 The Main Program Loop . The main program loop consists of the MAKEINPUT module, the PROPAGATE module, the TRAINNET module, the TESTNET and the DISPLAY and SHOW modules. A counter was used to inhibit the calling of some modules through the loop to reduce computations. For example,

the screen may be updated only every few hundred cycles. Each pass through the loop represents one training cycle, so MAKEINPUT, PROPAGATE, and TRAINNET will always be called.

2.5.4.1 Make an Input Vector and Desired Output Vector. A set of input patterns and a defined classification, are essential to the training of a neural net. In other words, an input vector and a desired output (doft) are needed for every update cycle.

2.5.4.2 File Input of Examplar Sets. Most problems can be described in terms of a set of input vectors and a defined classification. An exemplar is selected randomly from the pool whenever the MAKEINPUT routine is called. The routine should ensure that in addition to random selection of an exemplar number, there is also random selection based on class type. This prevents excessive training on a single class, when the classes are not evenly distributed in the input file.

2.5.4.3 Propagating the Input Vector . With the organization of the net in memory, the net can begin to learn and classify data. To use the net, the rules for propagating the data from the input to the output must be specified. This is the purpose of the Propagate package.

The feedforward routine is really only a hook to the real feedforward method, pointed to in the *layer* data structure. The function pointed to in the *layer* data structure only update one node. So, the feedforward routine will loop through all the nodes.

2.5.4.4 Updating the Weights. The TRAINNET module specifies the training algorithm for the network. It is only a hook to the real update routine. Because the update of a particular node may depend on all nodes in a layer, the update routine will update an entire layers weights.

2.5.4.5 Measuring the Error. Network performance is evaluated in two ways. First, training is periodically stopped and a test set is evaluated. The second method checks the performance after every training cycle against the current training vector. For a general evaluation of the neural net performance, a set of training data is run through the net without training cycles between tests. For a more specific analysis, using a data set different than the one used for training, can show the validity of the feature set used to classify the targets. The TEST routine allows this type of checking mid-process by running a quick test set through the net, then measuring the performance. The test set is specified in the initialization routine. When the data is read into memory during initialization, the first line of the file specifies the number of training exemplars followed by the number of test exemplars. This partitioning of the data allows the TEST routines to test the net with a set of vectors the net has not seen before.

2.5.4.6 Displaying the progress. The display routine refers only to the graphics portions of the display. This includes such functions as drawing the network weights, drawing the nodes, setting the colors, finding data ranges, and drawing color bars. The package is split into two types of graphics routines. The basic set contains those functions that are machine dependent. In general, a macro is used when possible to allow for redefinition to other machines. Those which are not machine dependent are combinations of the basic routines that are machine dependent. An example would be the color bar routine. These two types of routines are included in the graphics package. In general, when the problem under consideration changes from something like a Kohonen map to a counterpropagation model, the entire display package is replaced.

2.5.4.7 Interactive Program Control. An event-driven menu provides control for housekeeping functions of the network. Event driven menus require a hardware event to call the menu subroutine. No device polling is necessary. The

event, in this case, is typing a control C on the keyboard and is detected using **signal.h** . A control C activates a hardware interrupt to kill address vector. The program has substituted the normal kill address vector with the menu address vector.

While the main purpose of the menu is to allow user to save and restore weights, the menu also allows control parameters to be changed while training is in progress. A menu display in a text window offers a series of selections. Selection of a particular item will then prompt the user of the parameters associated with the particular call.

2.6 Summary

This chapter consisted of a literature review in the areas of objected-oriented design, graphical user interface, X Window System, and the current status of *Neural-Graphics* software package. In the first section, Object-oriented design was defined and various concepts described. The key terms in object-oriented techniques are object and class. An object is something which can be changed. It has behavior, state, and identity. When objects have a similar structure and behavior they are often grouped into classes. Another term defined was inheritance. C language, which was used to implement ANNS, does not support inheritance. Two object-oriented design processes were described, including an in-depth description of Booch's process. The research conducted showed that the process, as described by numerous authors, is basically the same. The first step is to identify the objects and group them into classes. At the same time as objects are identified, the operations which those objects require can be determined. As with other design techniques, an object-oriented design process should be an iterative one. This section also described some of the advantages and strong points of object-oriented design. Object-oriented design techniques provide the implementer with an easy way to follow sound software engineering principles. Object-oriented design techniques provide a modularized system which permits easier maintenance of the actual code. A good object-oriented

design ensures weak coupling and strong cohesion as well as supported abstraction. These are all very important software engineering principles.

The review of graphical user interfaces identified some of the desirable qualities of user interfaces. This section discussed "how" a user interface should be designed so that the users will feel comfortable with the system and can be more productive. Then, an overview of the X Window System and its extensions was presented. The X Window System is a tool user interface designers can use to construct professional, and hopefully, user-friendly interfaces. Lastly, the original software design and data structures of the *NeuralGraphics* package were described. This section outlined the major software modules which make up the *NeuralGraphics* environment. The following chapter will present the requirements analysis and specification for the development of the ANNS system.

III. Design Methodology, System Requirements Analysis and Preliminary Design

3.1 Introduction

This chapter presents system requirements and preliminary software design for ANNS - The AFIT Neural Network Simulator. Before work can proceed on any project an outline guiding the development of this work must be produced. The software design methodology of ANNS system is presented in the first section of this chapter.

3.2 Design Methodology.

An important part of any software development project is the overall model or methodology for accomplishing the task. The methodology outlines the steps to be taken from inception through implementation to retirement. It provides an organized approach to software development and allows for management of the development effort.

3.2.1 Current Methodologies. Currently, there is no one standardized methodology being used for software development. Unless a method has the flexibility, it is doubtful that any particular one will be perfect for all software development. However, some organizations have adopted one method over the others and tend to force all software development activities to follow the adopted mode. Some of the more popular methodologies in use today include:

- Classic Life Cycle (Waterfall)
- Evolutionary (Prototyping/Interactive)
- Program Transformation
- Spiral Model

The classic life cycle paradigm calls for a systematic approach to software development. The step in this approach include (see Figure 6) [42]:

- *Requirements analysis and definition.* The system's services, constraints, and goals are established and defined.
- *System and software design.* Using the requirements definition as a base, a design is drawn up of the system that can be readily transformed into a computer program.
- *Implementation and unit testing.* The software design is transformed into software units. These units are tested to verify that they meet the specifications.
- *System testing.* Units are integrated and tested as one complete unit. After the testing is completed and successful, the software is delivered to the user.
- *Operation and maintenance.* The software is installed and put into use. Maintenance occurs throughout the system lifecycle.

In actual practice these phases are rarely distinct; they can overlap and feed off each other. A major problem with this approach is that the system development rarely follows the sequential flow. Iteration always occurs and creates problems because it is difficult to determine the project's progress. Another major problem is that it is often difficult for the user to state all the requirements up front. Therefore the general consensus was that the Classic Life Cycle model does not lend to itself well to the design of user interface system. This is particularly due to the limited dialogue between developer and user once the design starts. On the other hand, prototyping provides a bridge to communicate with the user to better determine the system requirements and to help prevent designing the wrong system.

The prototyping paradigm can be used when the user does not have a complete set of requirements. An initial system, which may lack any support processing, is developed by the designers to demonstrate to the user that the project is feasible. This initial system can take three forms [29]:

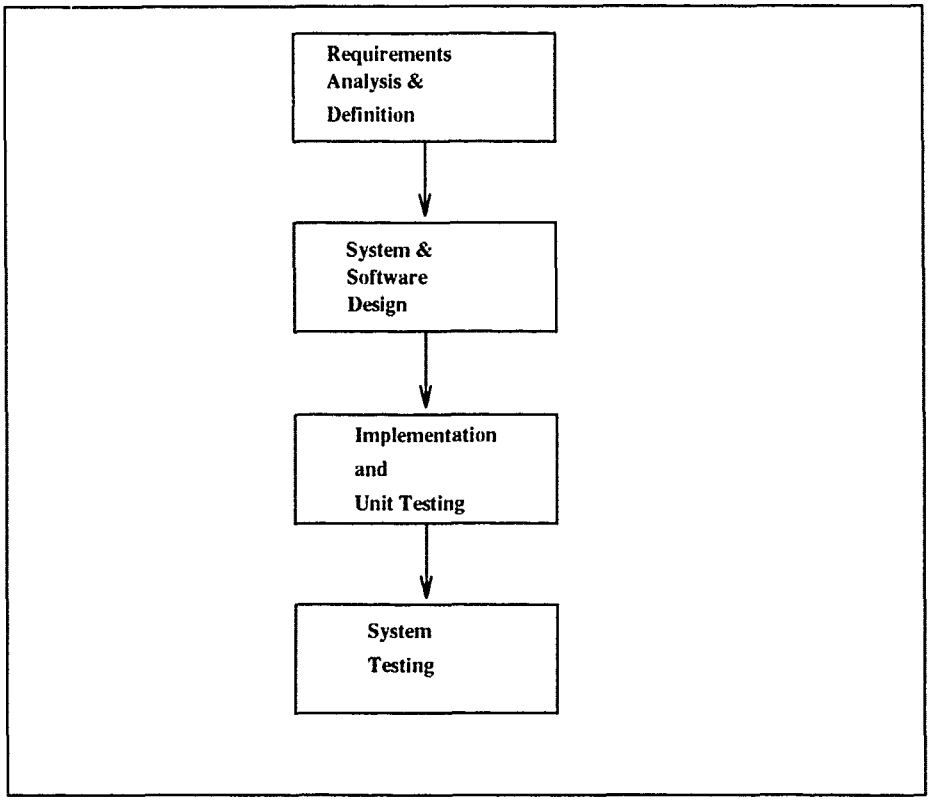


Figure 6. The Classic Life Cycle Model

- A paper prototype that depicts human-machine interface in a form that user can understand.
- A working prototype that implements some subset of the system.
- An existing program that emulates part of all of the functions desired but needs to be improved upon for this development effort.

This approach, like all others, starts with requirements gathering. After this initial phase, a *quick design* follows which focuses on those aspects of the system that will be visible to the user. This quick design leads to the development of a prototype that is evaluated and refined; a process of iteration which continues until the system fits user requirements. For the ANNS system development using this model, there are some existing programs that emulate part of all of the simulation functions desired but needs to be improved, such as neural network algorithm functions in *Neural-Graphics* system and X window graphical user interface protocol in some existing simulation system. However, prototyping does have some problems such as reliability, robustness, and safety that cannot be adequately expressed. Still, the ANNS graphical user interface is a prime candidate for prototyping, because it depends on visual displays and heavy interaction with the user [29].

The Iterative design paradigm has the primary advantage that a working system is produced at each iteration. Thus, user interface capabilities and improvements can be incrementally added to the system. The key point to this methodology is that the system is broken down into small manageable pieces, which are prioritized, and implemented one at a time. This is attractive for this effort because of the time constraints involved. A workable neural network simulation subsystem and graphical user interface are developed at each stage and are ready for implementation.

The Program Transformation methodology is emerging as an attractive alternative to program generation. In this paradigm, a formal specification of the system requirements is produced. This formal specification is then automatically trans-

formed into syntactically correct code. Some human intervention may be required to assist in the transformation. The generated code is then validated against the user's requirements. If the system must be modified, then the adjustments are made to the formal specifications and the process is repeated. With this method, there is no design stage. Since a large part of user interface generation is developing the screen layout, this system is not the most desirable. This is because many graphical screen designs can be produced from the same set of requirements.

The Spiral Model is a risk-driven approach to the software development process [1]. In this methodology, the following steps are repeated until the program is fully developed.

1. Determine objectives, alternatives, and constraints.
2. Evaluate alternatives.
3. Identify and resolve risks.
4. Develop and verify the next-level product.
5. Plan next phases.

Depending on the identified risks, the fourth step listed above may use the classic life cycle, prototype, iterative, or transform approach. Barry Boehm writes [1:65]:

> The spiral model also accommodates any appropriate mixture of a specification-oriented, prototype-oriented, simulation-oriented, automatic transformation-oriented, or other approach to software development, where the appropriate mixed strategy is chosen by considering the relative magnitude of the program risks, and the relative effectiveness of the various techniques in resolving the risks.

3.2.2 The Methodology Decision. It is sometimes the case that no one methodology is best for a software design project. For ANNS system development,

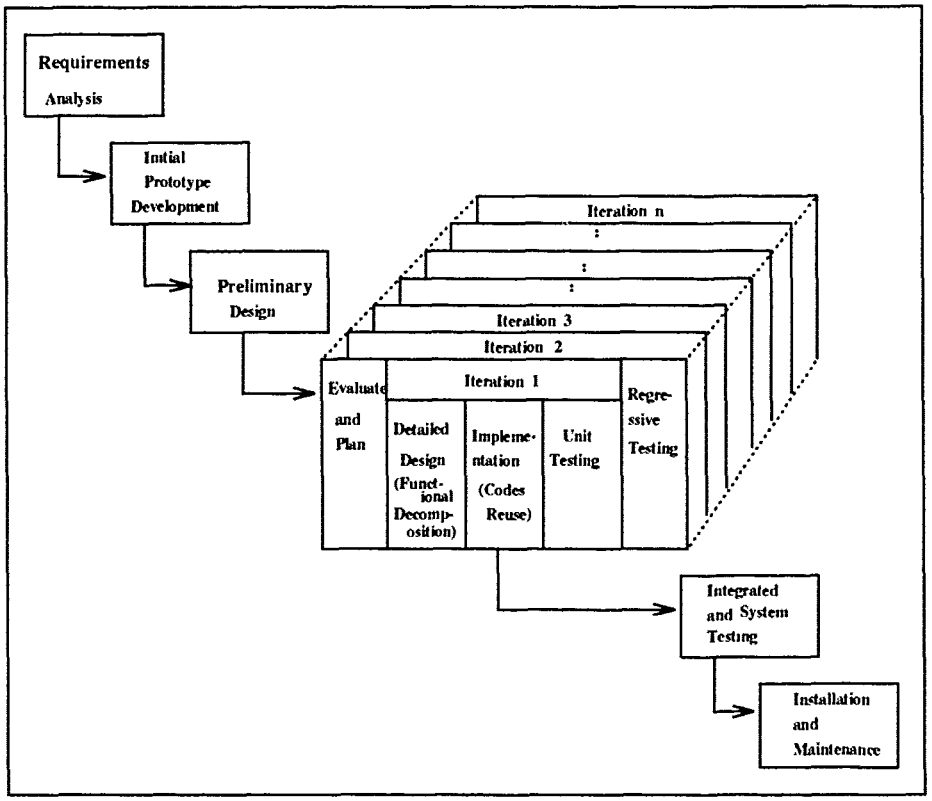


Figure 7. ANNS Design Methodology

a hybrid paradigm which combines the best characteristics of the classic life cycle, prototype, and iterative methodologies was used (as figure 7).

The first step involves an explicit requirement analysis phase before developing the initial prototype. Before jumping into a software design, the software engineer should have an understanding of at least the fundamental terms and concepts of the problem domain. If the software engineer has *current* experience in the domain area, this step may be omitted. However, if the designer has little or no experience or if recent advances have occurred in the field, a study or review of current topics should be conducted. Accordingly, the domain analysis conducted for this ANNS system consisted of:

- Reviewing recent technical reports in the areas of neural network simulator.
- Reviewing recent thesis efforts in the areas of designing graphical user interface.
- Conducting a literature review in the areas of algorithm animation and X window system.

After conducting the domain analysis, initial prototypes were developed. These considered of sample screen layouts to show the expected behavior of the ANNS system. The prototype stage was followed by an object-oriented high-level design of the overall user interface.

After finishing the preliminary design and before conducting the detailed design for each iteration, an evaluation and planning stage was conducted. This consisted of evaluating various alternatives for implementing the next portion of design. For each alternative, the expected benefits and risks were weighed. In addition, any constraints placed on the user interface were considered for possible impact. A plan was then developed for implementing the selected alternative in a manner that would maintain or enhance the object-orientedness of the overall design. Efforts were made to reuse as many components as possible while maintaining high cohesion and low coupling between modules and iterations. Using the appropriate plan and functional

decomposition, a detailed design was created. This, in turn, was used for generating the code for the given iteration. Unit testing was then performed on the individual module to ensure the proper functionality was achieved. Any time a major change or modification is made to a piece of software, there is the possibility of introducing errors into previously correct code. Thus, unit testing was followed by regression testing to ensure the newly added module did not adversely affect the functionality of previously written modules.

After the last iteration was completed, the end-user and client-programmer user interface software were integrated with neural network simulation software to form ANNS system on the Sun workstations.

3.3 System Requirements Analysis.

While the design model provides the 'How' for ANNS system development, the requirements analysis gives the development the 'What'. This section presents the system requirements analysis for ANNS. Since the goal of this effort is to establish an integrated and unified neural network simulator system, using the ANNS system overview model (Figure 8 is an idealistic abstract model) as a starting point, the analysis is conducted from two points of view: the end-user and the client-programmer of the ANNS system.

Based on the results of the literature review, the structured analysis methodology of Edward Yourdon [60] was used and the enumerated requirements specification were compiled.

The idea behind structured analysis is to reduce the complexity of a problem by hierarchically decomposing the problem into pieces that can be more easily understood. The decomposition can be based on data or processes (methods) and is reflected through a series of *function diagrams*. Each functional diagram illustrates one level of the decomposition. Figure 9 shows the highest-level object functional diagram for ANNS.

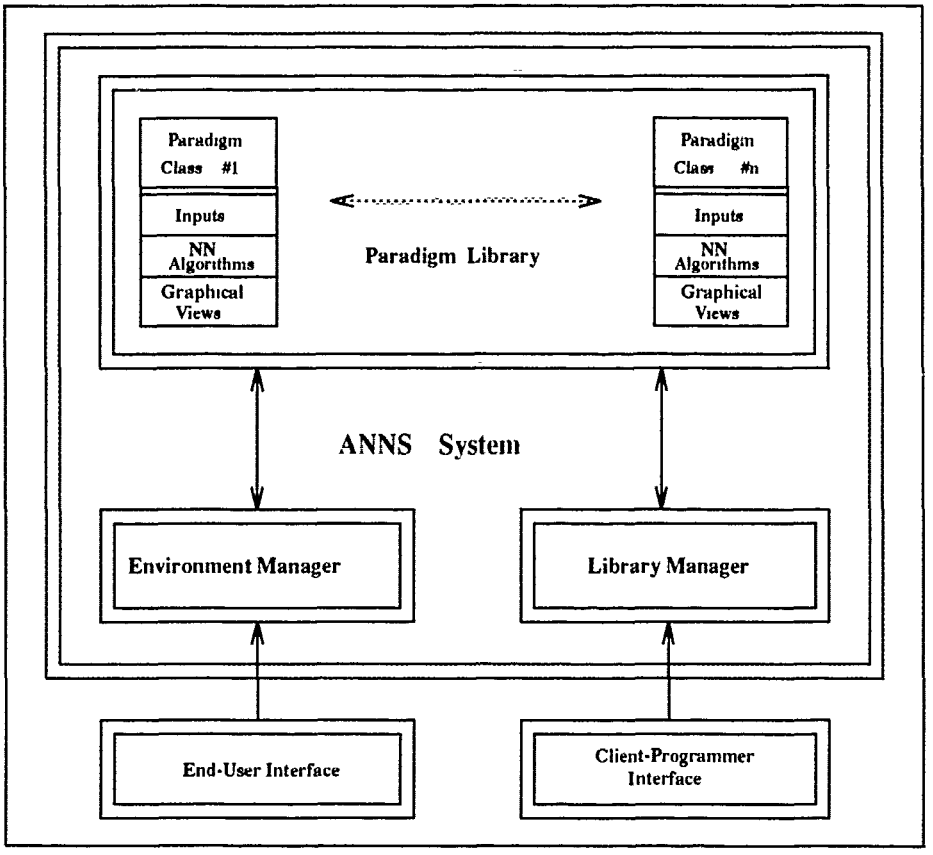


Figure 8. The ANNS System Overview

A process is presented by a circle on the functional diagram. The process name and number appear in the circle. The process number provides a means for tracing through the hierarchical decomposition. Interfaces between processes are represented by arrows entering or leaving process circle. The arrows represent data produced by or needed by a process.

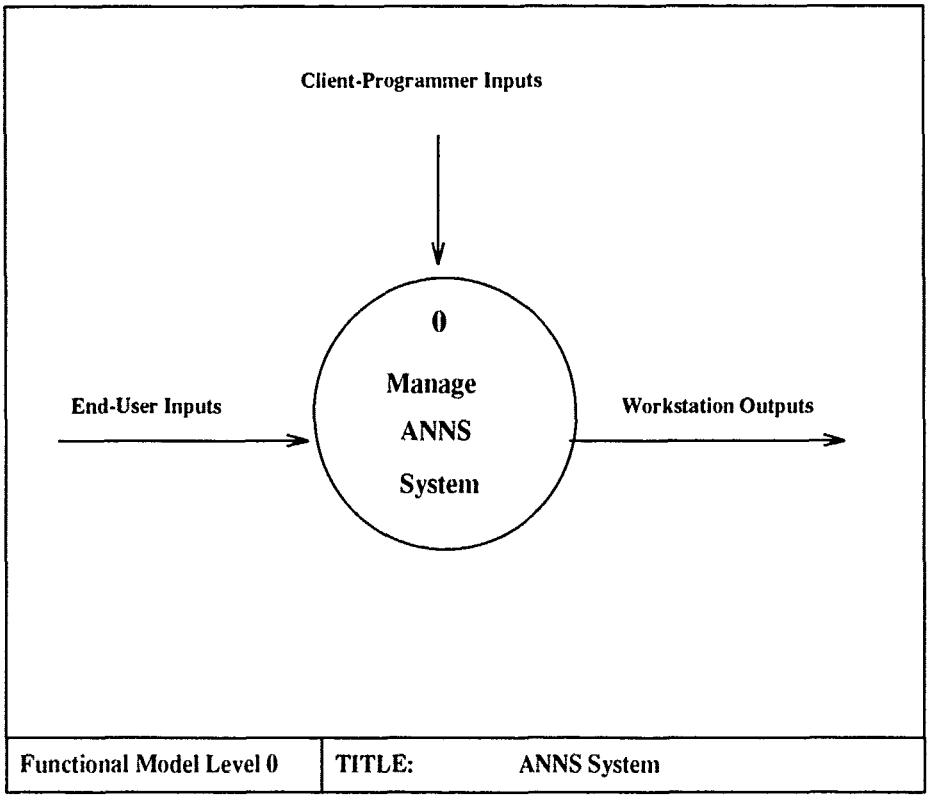


Figure 9. The Top-Level Object Functional Model of Design for ANNS System

3.3.1 End-User Requirements Analysis. The end-user of ANNS system should be able to customize the simulation environment. Multiple Neural Network (NN) algorithm windows, with multiple views of the neural network algorithm in execution should be possible. The user should control the position and size of NN algorithm windows and views. The user can close NN algorithm windows at any time. Likewise, a neural network algorithm window's contents may be replaced at any time.

The user may control each simulation window individually or control all windows simultaneously. In either case, controls available to the user are: start or continue a simulation, restart or reset a simulation, pause or terminate the simulation, single-step the simulation, and control the speed of the simulation. The simulation system may be exited at any time.

Neural network algorithm windows must provide a mechanism for monitoring and modifying the input parameters. In addition to the view windows, every neural network algorithm window should support a *status display* which presents statistics describing the current state of the execution.

The neural network simulation should be very flexible for the user to save and restore a particular environment, where the environment includes all user-selectable options available at a particular time, such as position and size of windows, views, neural network algorithms, and input parameters.

To help users understand all the options available, on-line help should be available for every interactive function.

3.3.2 Client-Programmer Requirements Analysis. A well-defined, consistent programmer interface is essential to the effectiveness of a neural network simulator. If new neural network algorithms or paradigms, views, and input parameters cannot easily be added to the system, the system is not an integrated simulator system and cannot fulfill its purpose. The client-programmer should be able to create, modify, and delete paradigms or algorithms within the neural network simulator system.

The system should provide a library of all neural network algorithm simulations as well as a library of primitives to support color visualization. The system should be structured such that the neural network algorithm simulations are independent and separate from the main simulation controller. Individual neural network algorithm

simulations can be added, deleted, and modified from ANNS with no effect on other neural network algorithm simulations or the system as whole.

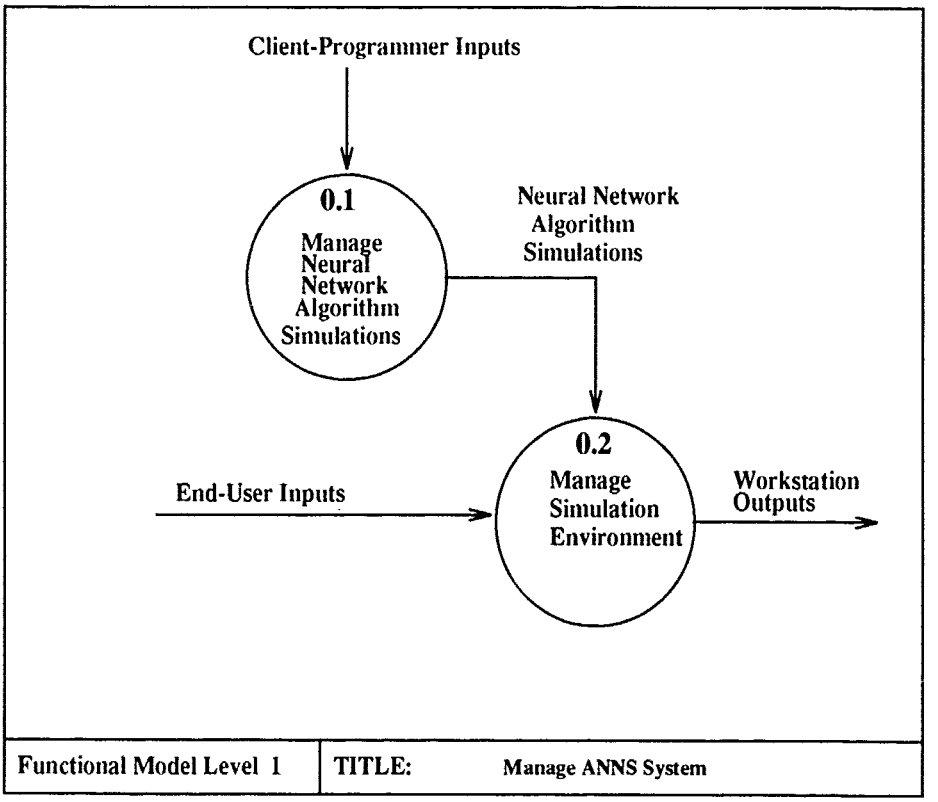


Figure 10. The Functional Decomposition Diagram Level 1: Manage ANNS System

3.3.3 Enumerated Requirements for ANNS System. (see Figure 9, 10, 11, 12, 13, 14)

1. Establish a user interface to ANNS environment manager

   (a) Allow user to customize the ANNS environment

      i. Provide multiple neural network algorithm windows

      ii. Provide multiple graphical view windows within each neural network algorithm window

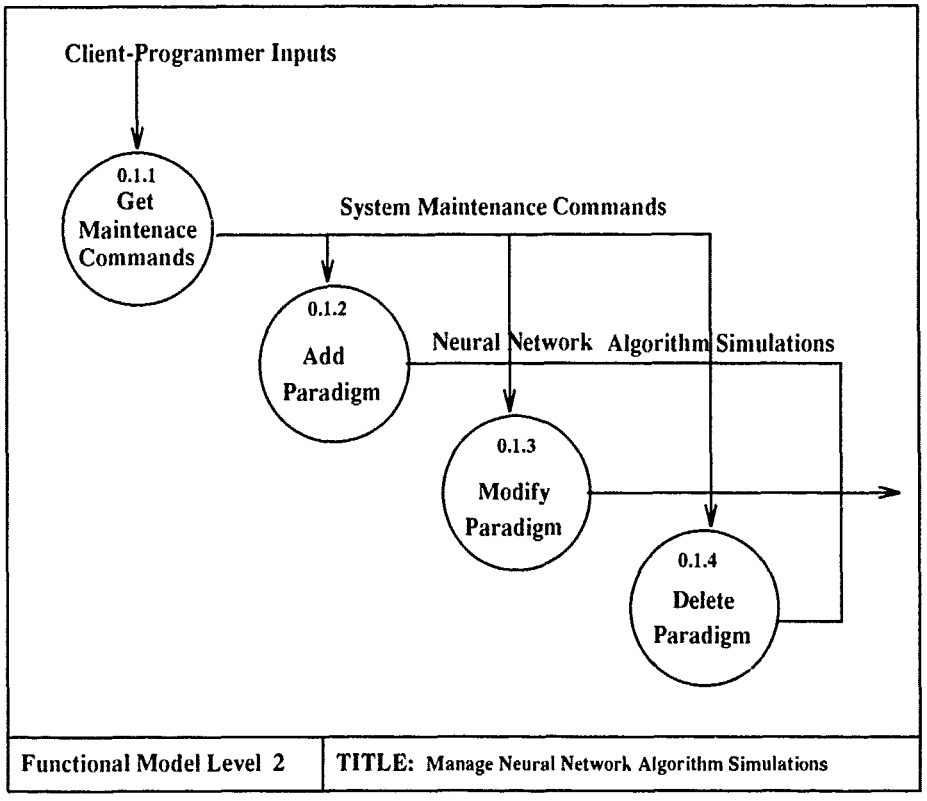


Figure 11. The Functional Decomposition Diagram Level 2: Manage NN Algorithm Simulation

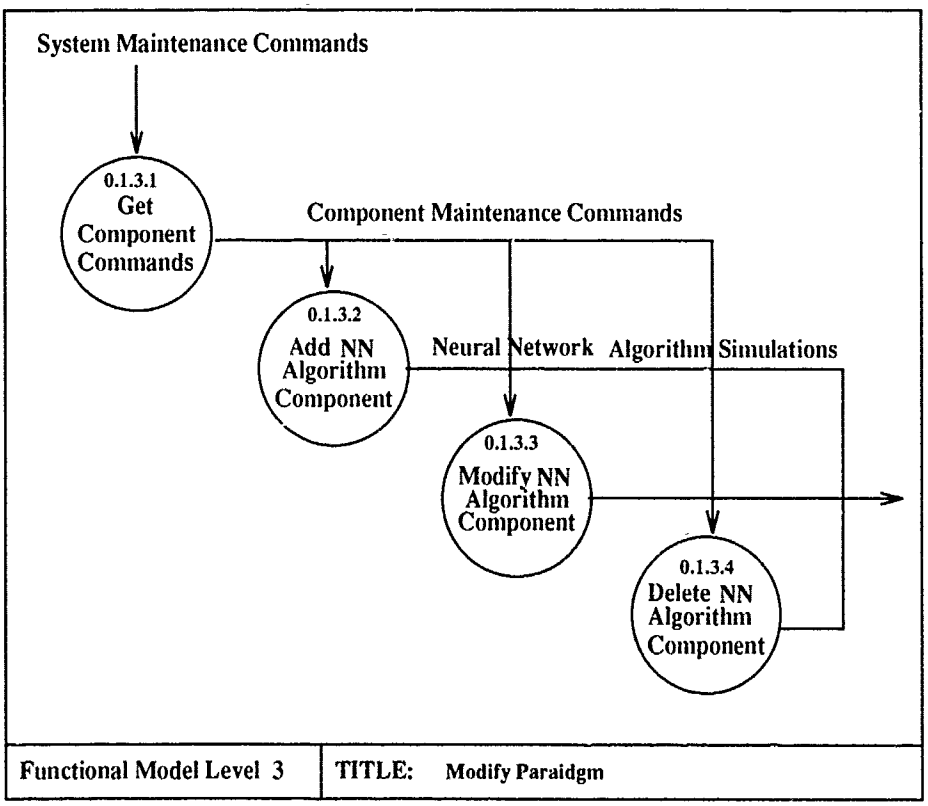


Figure 12. The Functional Decomposition Diagram Level 3: Modify Paradigm

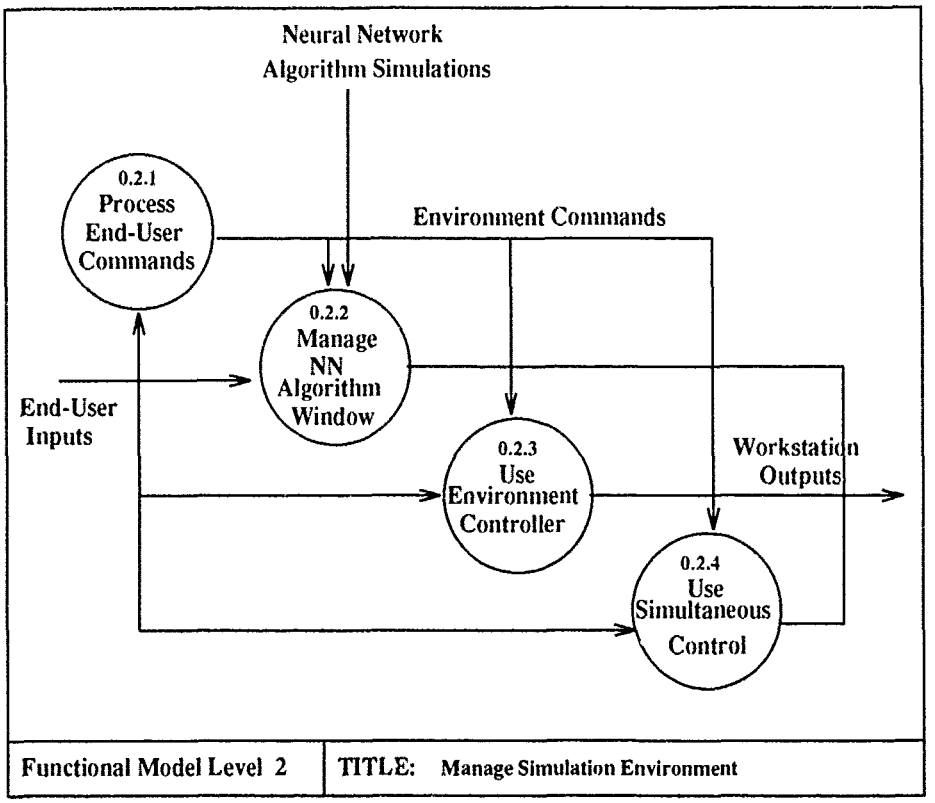


Figure 13. The Functional Decomposition Diagram Level 2: Manage Simulation Environment

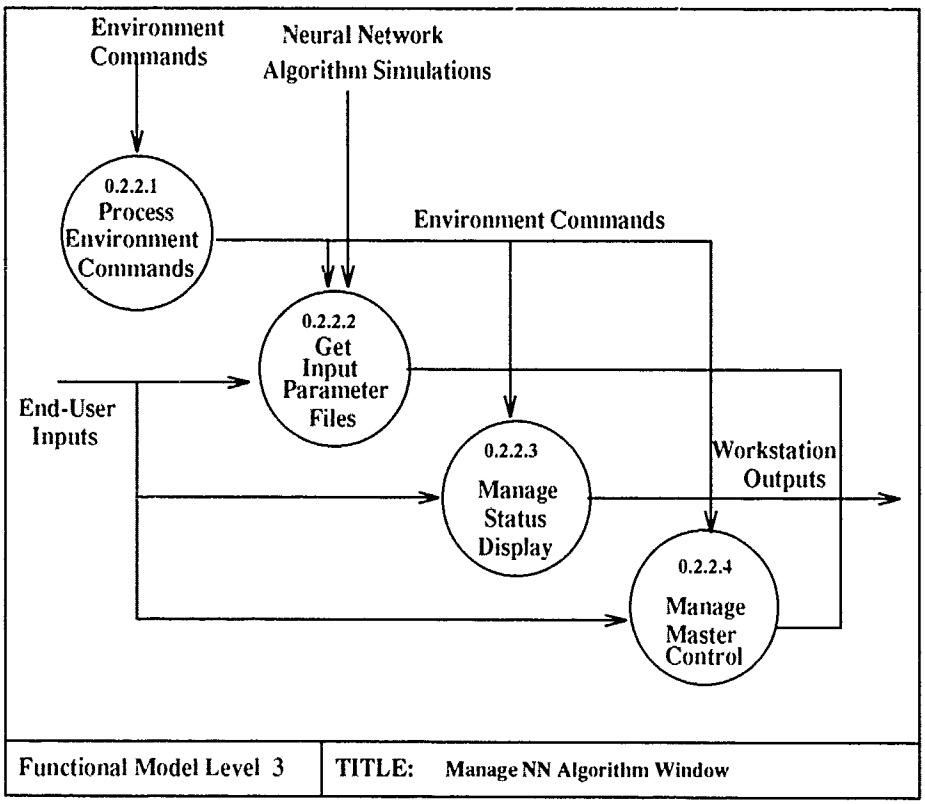


Figure 11. The Functional Decomposition Diagram Level 3: Manage NN Algorithm Window

iii. Allow user to position and size neural network algorithm and graphical view windows as desired

iv. Allow user to zoom and pan display in view windows at any time

v. Allow user to terminate simulation session at any time

(b) Allow user to select paradigms, neural network algorithms, and input parameters for each neural network algorithm window

i. Select paradigm class from a list of available classes stored in the paradigm library directory, such as *back-propagation, hybrid training, radial basis function, hopfield associative memory, etc.*.

ii. Select neural network algorithms for simulation from a list of available neural network algorithms within a selected paradigm class.

iii. Specify input parameters for a neural network algorithm from a list of input parameters associated with a selected NN paradigm

iv. Close simulation window at any time

(c) Allow user to control execution of neural network algorithm simulations

i. Select control mode: individual or simultaneous

ii. Start or reset a simulation

iii. Control speed of simulation

iv. Terminate or pause simulation at any time

v. Restart a paused or terminated simulation

vi. Run simulation in a single-step mode

(d) Provide a simulation control mode to allow user to control all the simulations appearing at the same time

(e) Provide environment save and restore function

i. Save all user selections currently in effect

ii. Restore previously saved environment

(f) Provide on-line help for every interactive function

2. Establish a neural network programmer interface to neural network paradigm simulation library

   (a) Allow neural network programmer to create, modify, and delete neural network algorithm simulation within ANNS system

      i. Add, modify, and delete neural network algorithms from neural network paradigm class.

      ii. Add, modify, and delete neural network algorithm graphical views

      iii. Add, modify, and delete neural network algorithm input parameters

   (b) Provide automatic validation of changes to ANNS system

      i. Interface errors are reported immediately

      ii. Protect ANNS system from accidental corruption during development of new neural network simulations

   (c) Provide library of primitive functions to support coloring simulation

3.4 Preliminary Design

After the requirements were determined, paper designs of the screen layouts were drawn and evaluated. These served as the starting points for the prototypes. Even though the prototypes did not have the full processing capabilities, they were useful in evaluating important aspects of ANNS system. This section identifies several design issues in developing these prototypes and the decisions made.

An *object-oriented* design methodology was used to translate the requirements into high-level *objects* and *operations*. The first step in producing the object-oriented design was to identify the major objects and object classes. Booch defines a class as a set of objects that share a common structure and common behavior [5:93]. For ANNS system design, two high-level graphical object classes and a neural network algorithm

component object class were defined. Figure 15 shows the high-level object classes and their subclasses. In this section, the high-level classes and their interfaces are defined. Specific object instances and their relationships are also described.

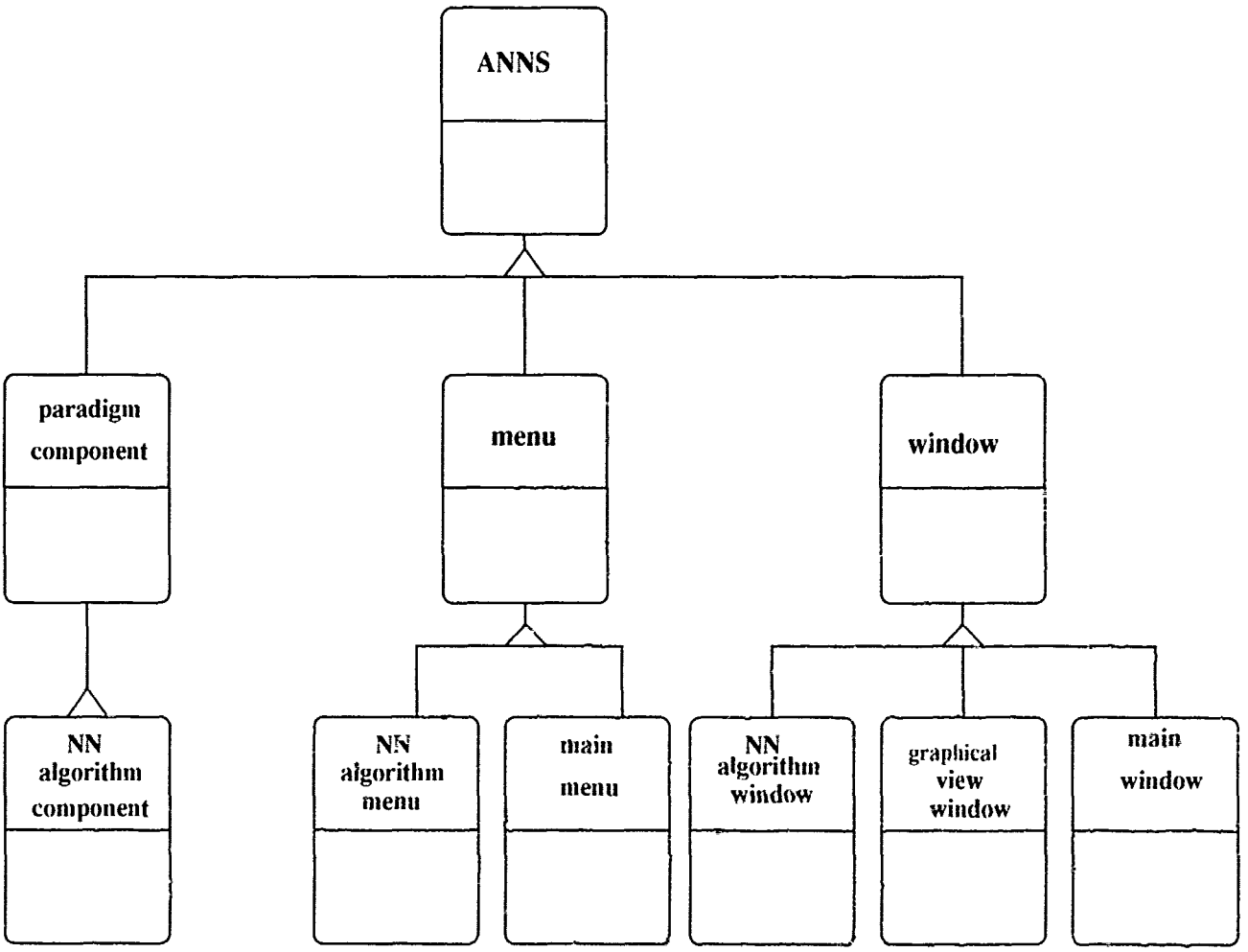


Figure 15. High-Level Object Class Structure

3.4.0.1 The **Window** Object Class. A window is an abstraction that represents a rectangular region of the workstation display. Windows can be moved and resized. Text or graphics can be displayed in a window. Windows can be displayed within other **windows** . Figure 16 shows the abstract data type (ADT) specification for the **window** object class.

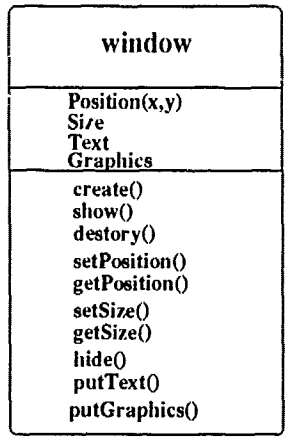


Figure 16. ADT Specification for ANNS **window** Object

The ANNS design incorporates three levels of windows to provide the user with multiple simultaneous neural network algorithm simulation and multiple graphical views of each simulation. Figure 17 shows the instances of window used by ANNS and their relationship to one another.

- **Main Window.**

  This is the highest-level manager window of the ANNS environment: all interaction with ANNS is contained. No simulation actually takes place in this window. The main window supports multiple instances of the *NN algorithm windows.*

- **NN Algorithm Window.**

  Like the main window, NN algorithm windows exist mainly to contain other windows. Every NN algorithm window is associated with a particular NN paradigm class. NN algorithm window may be created and destroyed at any time, but there is a limit to the number that can exist at any given time. Each NN algorithm window supports multiple instances of *graphical view windows.* NN algorithm windows can be resized and moved anywhere on the screen of the Sun Sparc2 workstations and they may overlap one another.

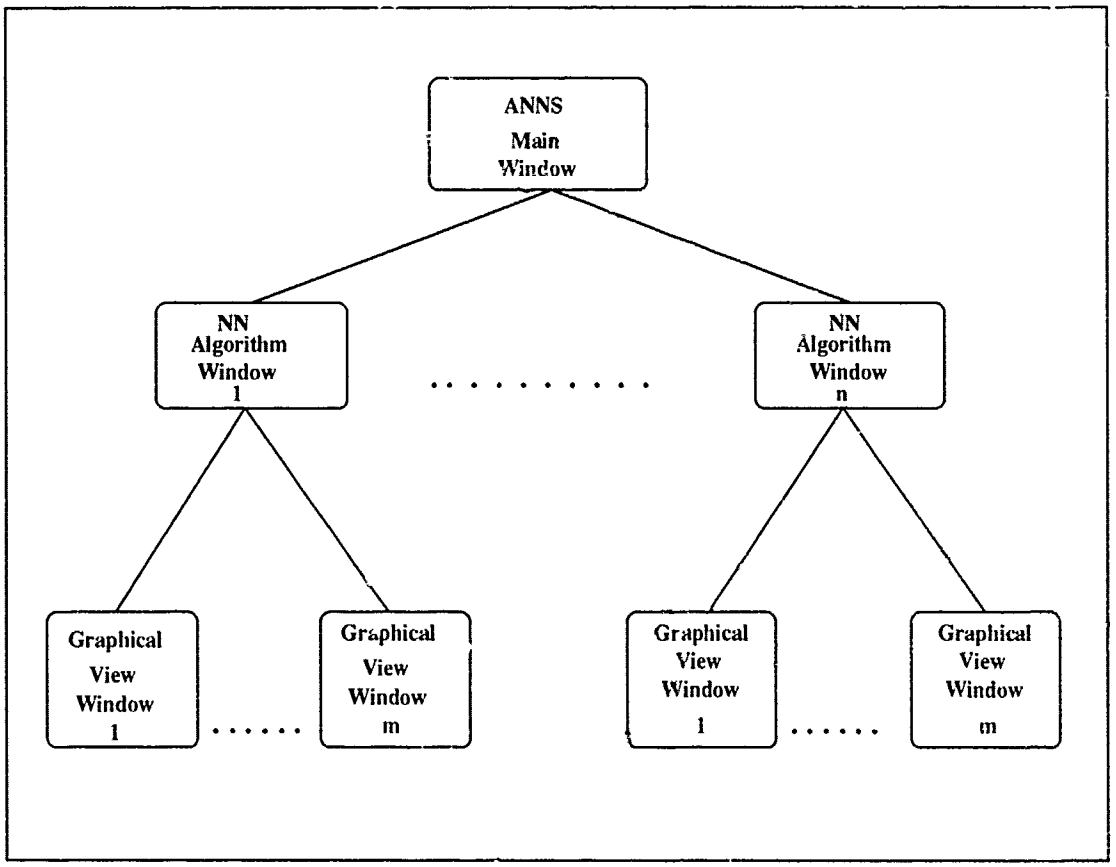


Figure 17. Instances of windows

- **Graphical View Window.**

  Neural network simulations are graphically displayed in graphical view windows. Every graphical view window is associated with a particular graphical view of a neural network algorithm. Graphical view windows can be resized and moved.

3.4.0.2 The **menu** Object Class. The **menu** provides a method for users to make a selection from several options. In XView protocols, **menu** consist of low-level objects called *menuItems*. *menuItem* can be associated with a particular operation which invoked when the *menuItem* is selected. Figure 18 describes the **menu** object class with its instances and operations.

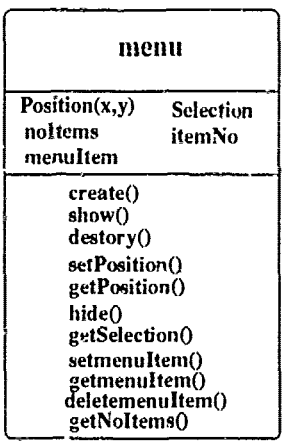


Figure 18. ADT Specification for ANNS **menu** Object

The ANNS provides two instances of **menu** for user with high-level decisions on two levels.

- **Main Menu.** This menu is used to select a new NN algorithm simulation and to active panels for simultaneous control of simulations and for saving and restoring simulation environments.
- **NN Algorithm Window Menu.** An NN algorithm window menu is associated with every NN algorithm window. This menu is used to activate various

panels which control the simulation, retrieve or save input data (weights) file and present a status display for the simulation.

3.4.0.3 The **component** Object Class. The **component** object provides means to select and control the NN paradigm components depicted in Figure 8. The component objects are associated with NN algorithm windows and graphical view windows, but not with the main window. Figure 19 shows the **component** object class with its instances and operations.

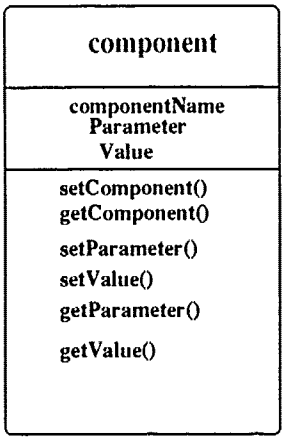


Figure 19. ADT Specification for ANNS **component** Object

The ANNS design uses three types of components: *input*, *NN paradigm*, and *graphical view*. Each type supplements the basic operations supplied by the class with a set of component-specific operations. Figure 20 illustrates how the **components** fit into the NN algorithm window structure.

3.5 Summary

This chapter presented a summary of design methodologies currently being used for software development. This was followed by a description of the hybrid methodology used for ANNS system. To start the first step of this hybrid software design model, the system requirement analysis was described in the form of functional

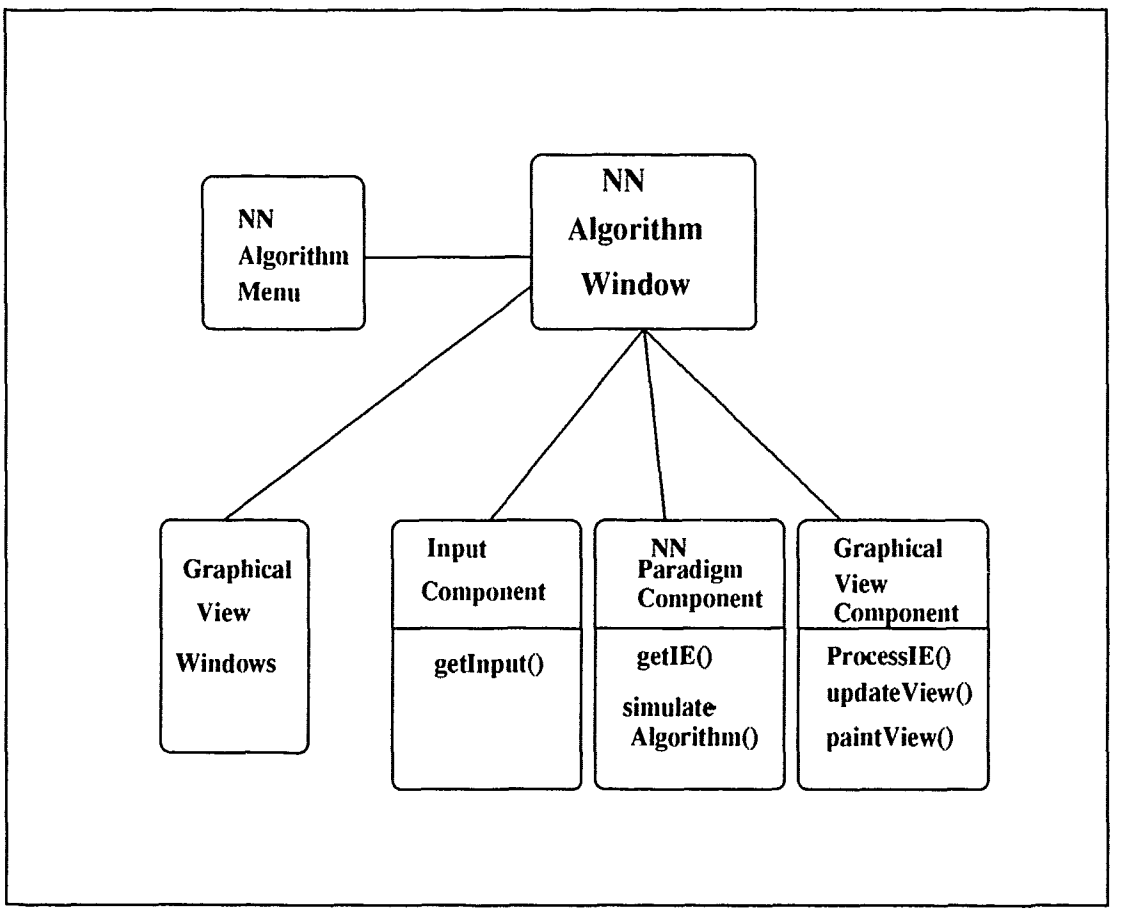


Figure 20. Structure of NN Algorithm Windows

decomposition and enumerated requirements. The object-oriented design technique and the design objects with operations were also described. In the next chapter, the preliminary design is translated into the detailed design and implementation of ANNS.

IV. Detailed Object-Oriented Design and Implementation

4.1 Introduction

This chapter describes the detailed object oriented design and implementation of ANNS on Sun SPARCstations using the XView window-based GUI (Graphical User Interface) environment. The purpose of the XView GUI is to bind all the other neural network subsystems into one homogeneous, user friendly environment. This implementation is based on the preliminary design developed in chapter III.

The topics discussed in this chapter include: the motivations for selecting XView as GUI, the detailed OOD design, the graphical representations of dependency modules, the implementation approach, the results of the implementation, and the testing methodology.

4.2 Motivations for Selecting XView as GUI

4.2.1 From Users Perspective. With the large influx of Sun SPARCstations, most contemporary users at AFIT are using OpenWindows simply because all of the Sun workstations currently in use at AFIT default to OpenWindows. The OpenWindows environment is largely implemented with the OPEN LOOK toolkit (parts of it are still in XView). OpenWindows is the default user interface environment for the SPARCstation2.[1] It is running on all SPARCstation2s at AFIT. The XView interface has much the same look and feel as the OpenWindows applications supplied by Sun. So, the goal, from a users perspective, is to make ANNS indistinguishable from other OpenWindows applications, which means end users do not have to learn another interface.

[1] This is an installation configuration decision. Sun bundles OpenWindows with the operating system. Of course, system administrators have the option of installing other environments

4.2.2 From Programmers Perspective. XView provides the programmer with a predefined set of interface components which are intended to simplify applications development. Many of the ANNS objects map directly to XView objects, significantly simplifying the implementation. XView is also an object-oriented and OPENLOOK compliant widget set from Sun Microsystems. The XView programmer's model was complied with the OPENLOOK standard. The XView widget set is not as sophisticated as that of other widget sets, like Motif and Athena which are also very popular in the X windows world. Of all these toolkits, XView has the easiest programmer model to understand and implement. A number of simple, but effective examples are provided with the OpenWindows development software and are the same examples used in [9].

4.3 Detailed Object-Oriented Design

In this section, the detailed design for each object that is part of ANNS is discussed. The physical design and object dependencies of ANNS is graphically depicted using *module diagrams*.

> A *module diagram* is used to show the allocation of classes and objects to modules in the physical design of a system; a single module diagram represents all or part of the module architecture of a system. [5:175]

The object dependencies for each detailed design module in ANNS was shown via diagrams composed of a modified version of module symbols presented in [1:55-59] and [5:175]. *Module diagrams* are used to represent the (hierarchical) procedural structure of the ANNS program modules. Module diagrams are an effective program structure notation, yet simple enough that practically anyone can understand them with little or no explanation. Figure 21 shows the three types of modules used in this chapter to describe the ANNS system packages. The first is used to represent the main ANNS subprograms. The second module represents the packages that encapsulate the object operations in the system. The third symbol is used as a space

saver to represent that there are several neural network subsystem which includes paradigm packages. The directed arrows between modules indicates compilation dependency where the module at the source of the arrow depends on the module at the destination of the arrow.

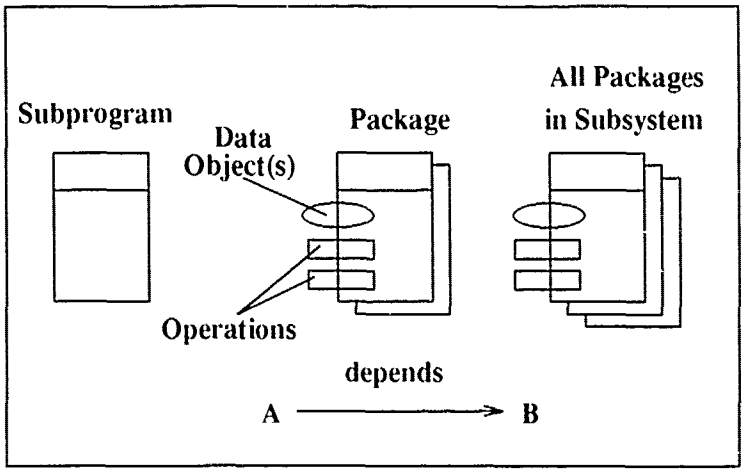


Figure 21. Booch Module Symbols

The module diagram of Figure 22 represents the architecture of the ANNS system. The design of the ANNS system consists of three levels of execution linked via UNIX sockets. Each class system is a self-contained, executable system. Without the sockets, each individual class system in the ANNS system could be run stand-alone. The three levels of execution are:

- *The ANNS main process.*

  The ANNS main process is the highest level interface to ANNS. It acts as the environment manager and creates the main process window, the main menu window, the central control window, the environment control window, the exit window, the help window and a means fo. starting NN paradigm simulations.

- *The ANNS common library.*

  The middle level of execution is the NN paradigm-specific window-based process called ANNS common library. Each NN paradigm component interfaces

with the common library for simulation control. The common library provides: the graphical view windows for viewing the dynamically graphical display, the master control window for monitoring and modifying the parameters that affect the simulation, and the online help window for understanding the use of all the parameters in the master control window.

- *The ANNS paradigms.*

  This is the lowest level of execution which is transparent to the users. Each NN paradigm has three windows associated with the ANNS common library. The graphical view window is directly under the paradigm's control. There is only one NN paradigm (multilayer feedforward networks using backpropagation) implemented so far, including eleven algorithms: Back Propagation, Back Prop W/Momentum, Second Order Learning, Cottrell Identity Net, Tarr/Cottrell Identity, Auto-Add a Layer, Gram_Schmidt Network, Gram-Schmidt ID Net, BrainMaker, Radial Basis, and Conic Basis. If there are time constraints for this thesis cycle efforts, the rest of the paradigms shown in the figure 22 could be implemented by client-programmers or by the next thesis cycles.

4.3.1 Main Process Window Module. The ANNS main process window is not associated with any particular NN algorithm simulation, but provides an environment in which any NN algorithm simulation can be controlled. Figure 23 shows the detailed design of main process window module.

4.3.2 Main Menu Module. The main menu module is for options to create NN paradigm menu window, central control window, environment control window, help window, and exit window. Figure 24 shows the detailed design module of main menu module.

4.3.3 Central Control Window Module. This module provides a means for simultaneously controlling multiple simulations. It creates HELP panel module, GO

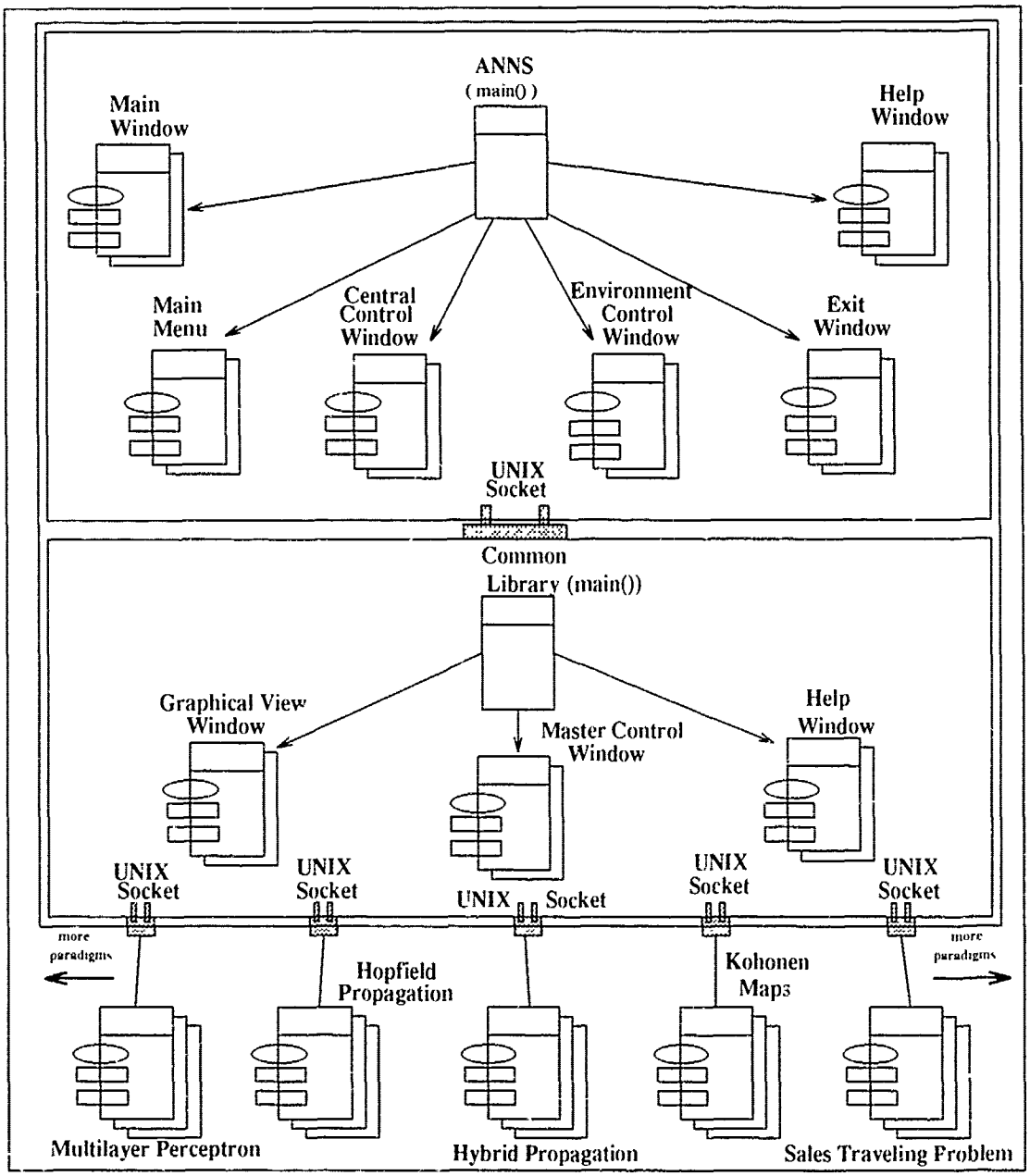


Figure 22. The Architecture of ANNS at the Top Level

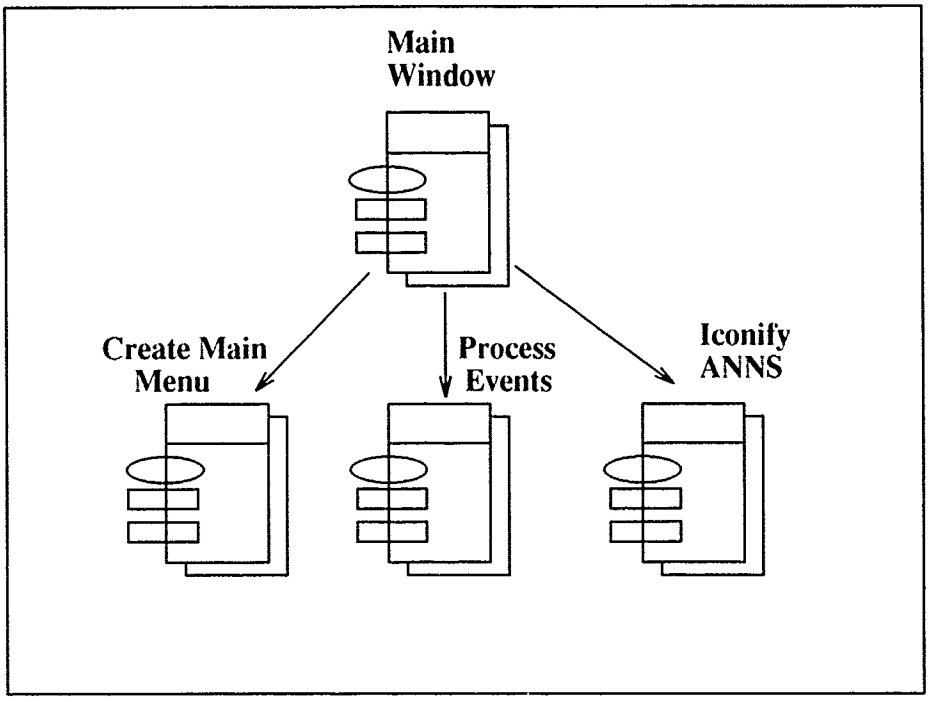


Figure 23. Module Diagram for Main Process Window

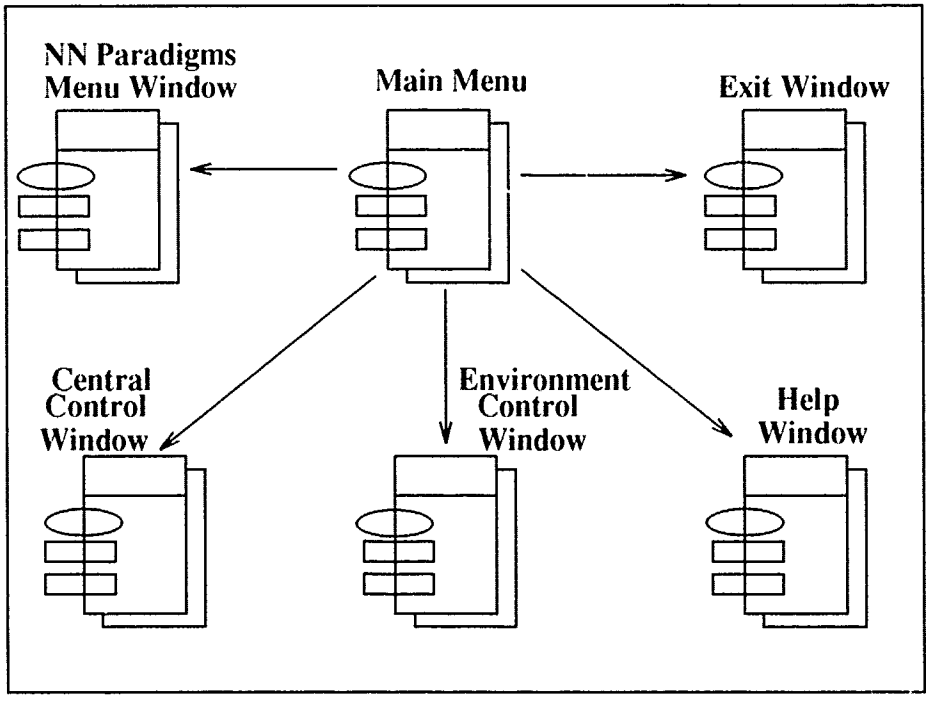


Figure 24. Module Diagram for Main Menu Module

panel module, STOP panel module, RESET panel module, and QUIT panel module. Figure 25 shows the detailed design of the central control window module.

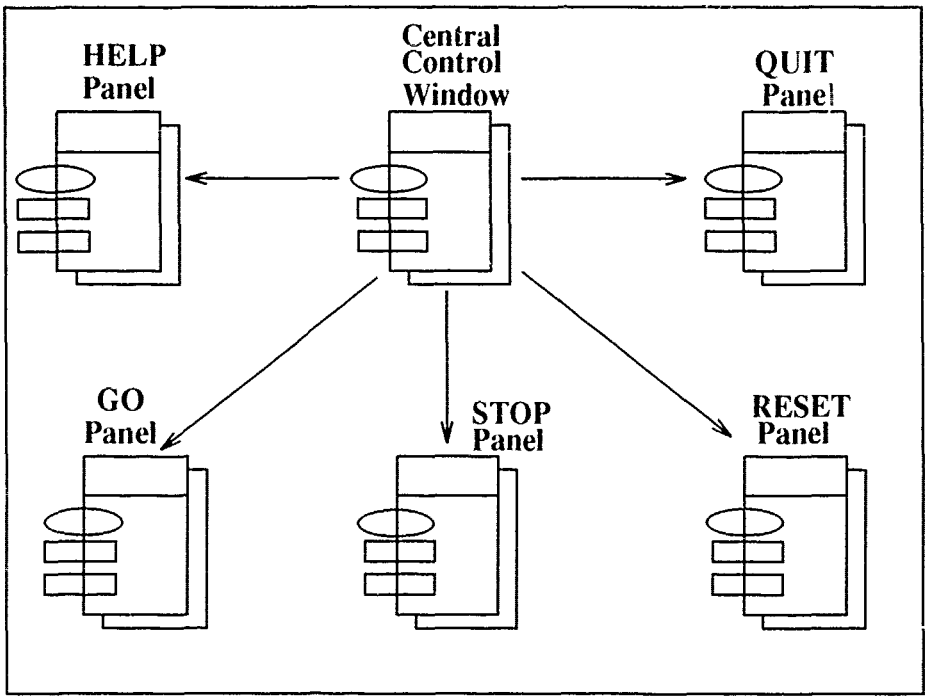


Figure 25. Module Diagram for Central Control Window Module

4.3.4 Environment Control Window Module. This module provides a means for saving the current simulation environment, restoring a saved environment, or killing a saved environment. The environment includes all users options currently selected: window size, window placement, and NN algorithm parameters. The environment control window module is shown in figure 26.

4.3.5 Exit Window Module. This module warns users that the executive status of ANNS would be killed and the execution would exit by providing two options to confirm exiting or cancelling. The Exit Window module is shown in figure 27.

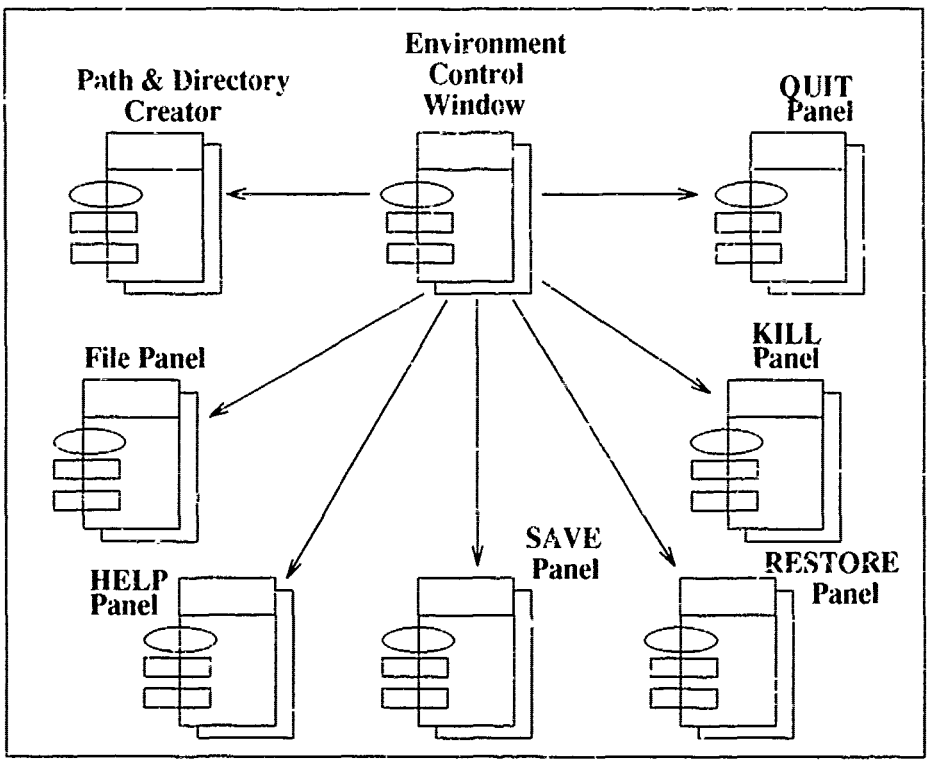


Figure 26. Module Diagram for Environment Control Window Module

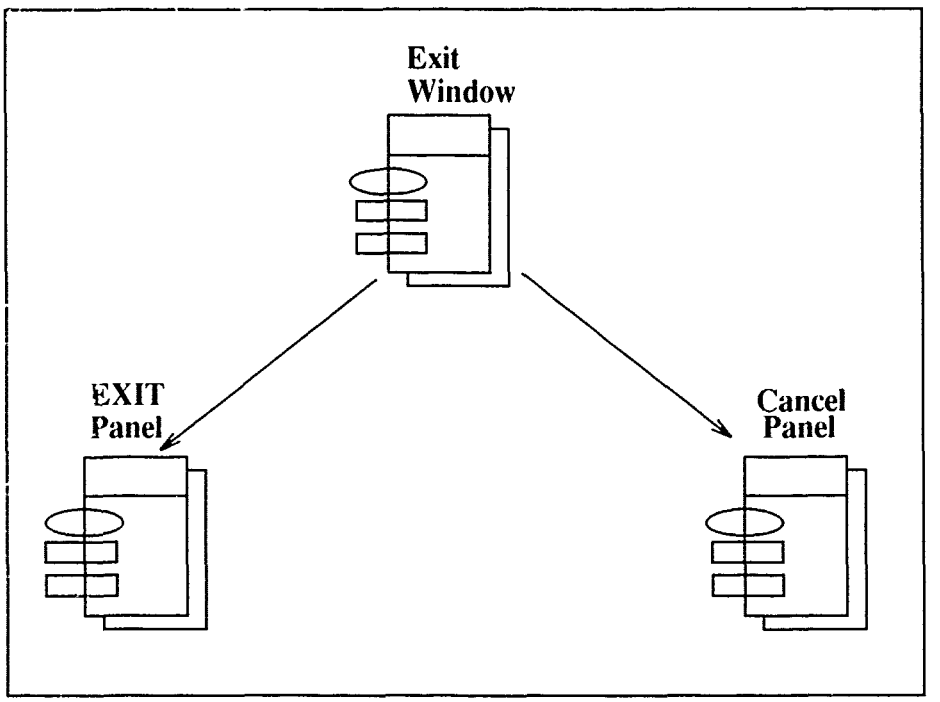


Figure 27. Module Diagram for Exit Window Module

4.3.6 Graphical View Window Module. This Module is the place NN simulations take place. It handles the NN simulation events and display the updated graphical view of NN algorithm simulation. The Graphical View Window Module is shown in figure 28.

4.3.7 Master Control Window Module. This module provides all options for users to control the NN algorithm simulation, such as the simulation speed controlling, "Go" simulation, "Stop" simulation, "Reset" all the parameters, and "Quit" the simulation. The Window is divided into several sections: Control Section, Algorithm Options, Simulation Status Display options, and Configuration options depending on the type of NN paradigms. Figure 29 shows the detailed design of Master Control Module.

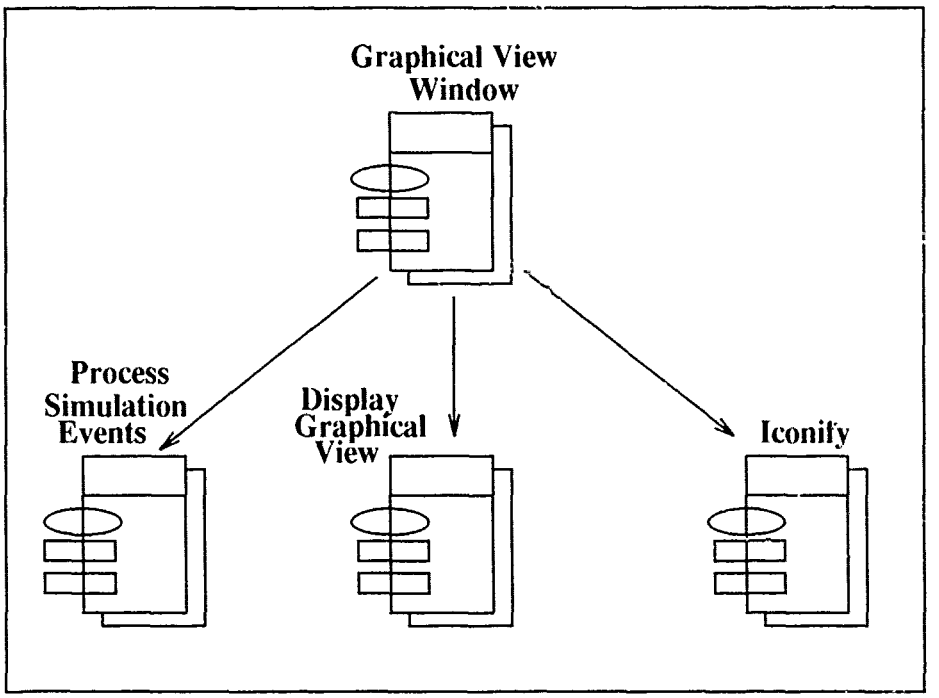


Figure 28. Module Diagram for Graphical View Window Module

4.3.8 Multilayer Perceptron Paradigm. This is one of independent NN subsystems in the third level of ANNS as in figure 22. It should provide the window-based level with the Interesting Events (IEs) that drive the simulation and update the graphical display on the graphical view window. The detailed design modules of this system is shown in figure 30.

Each subsystem or NN paradigm has one graphical view window and one master control window associated with it.

4.4 Implementation

In the implementation stage, C was selected as the programming language, since C provides the speed required for computer graphics intensive applications, and supports modern software engineering concepts. The XView and SunOS, Sun

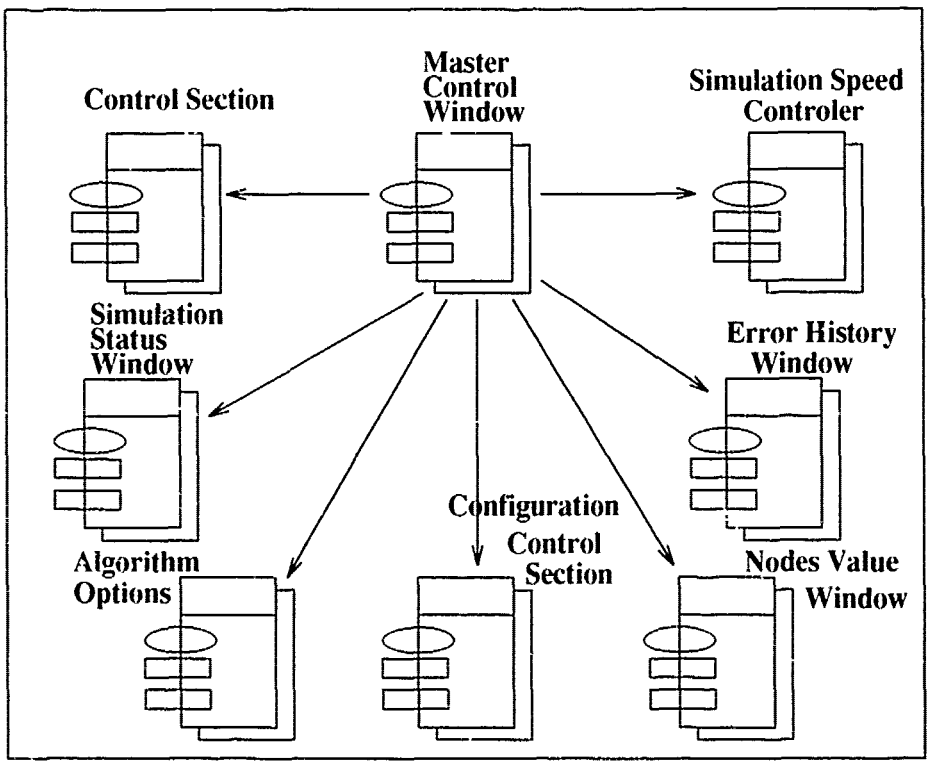


Figure 29. Module Diagram for Master Control Window Module

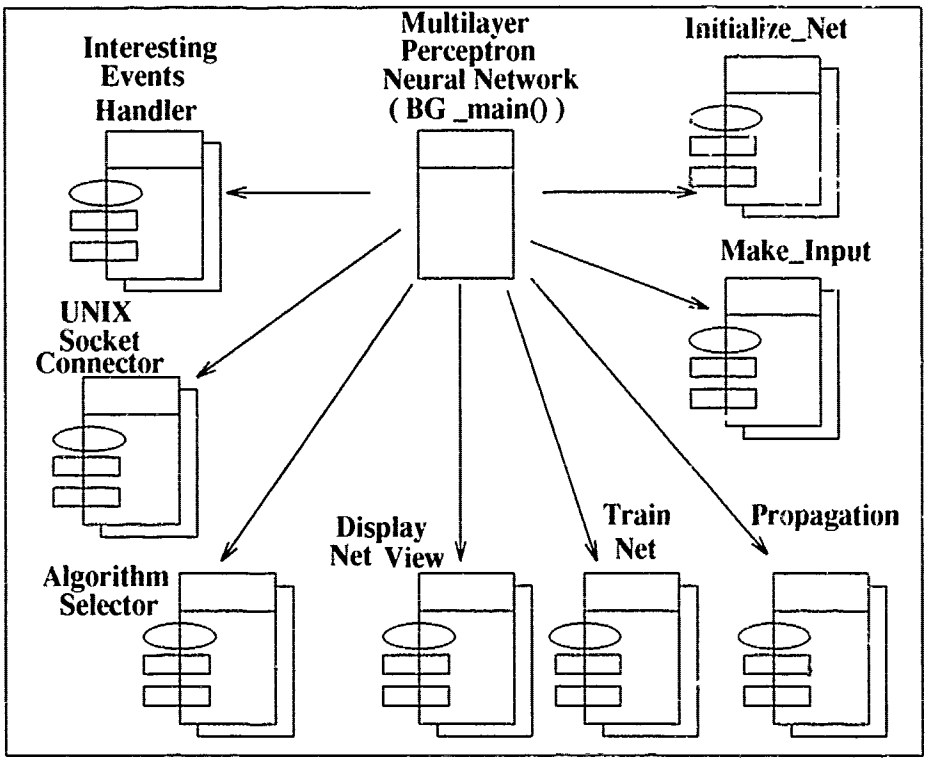


Figure 30. Module Diagram for Multilayer Perceptron NN subsystem

workstation operation system, provide an excellent interface to C programming language.

4.4.1 Creating and Mapping Objects from Detailed Design Modules. XView provides a very clean interface to it's object set. There is a common set of functions that allows users to manipulate any objects by referencing the object handle. The functions are:

- **xv_init()**:

  Establishes the connection to the server, initializes the Notifier and the Defaults/Resource-Manager database, loads the Server Resource Manager database, and parses any generic toolkit command line options. Called once at the beginning of the program.

- **xv_create()**:

  Creates an object. Every XView object is created with this function.

- **xv_destroy()**:

  Destroys an object.

- **xv_find()**:

  Searches for and returns an object with the specified parameters. If none is found, the object is created.

- **xv_get()**:

  Get the value of the specified attribute for the specified object.

- **xv_set()**:

  Set the value of the specified attribute for the specified object.

Using these six routines, programmers can create and manipulate the entire XView object set for ANNS.

4.4.2 Types of Objects in XView. There are eight basic object types in XView. Three of these, *Generic Objects*, *Windows*, and *Openwins* are core classes

and are not directly instantiable by the user. The remaining five are discussed below. The basic window entity is the *frame*. All other windows are classified as *subwindows* and must be attached to frames.

- **Frames**

  A frame is the basic window object to which the programmer has access. There are two flavors, a base frame, and a pop-up frame. A base frame is a frame with no parent. All other frames are subframes, so a pop-up is any frame which is not the base frame. Each application has one base frame. There are no (presets) limits on the number of subframes. A frame is characterized by a border, which is managed by the window manager, and an interior which is configured and managed by the programmer. The window manager controls resizing, iconification, de-iconification, refreshing, quitting, etc. All XView windows are framed.

- **Canvases**

  A *canvas subwindow* is the XView graphics window. It's size is independent of the owning frame. The entire drawing surface is called the *paint window* and the visible portion is the *view window*. Multiple, scrollable views of a canvas are allowed within a frame.

- **Text Windows**

  A *text subwindow* provides basic text editing capabilities using the OPEN-LOOK text editing model. This window is a specialization of the canvas subwindow with text editing capabilities added.

- **Menus**

  Menus are not actually XView windows, but they are bound to windows at display time. XView supports a full range of menu types and options such as pull-down, pop-up and pull-right. Menus can be *pinned* to allow them to stay on the screen after the selection is made.

- **Scrollbars**

  Scrollbars are interesting objects. They can exist independently, or be attached to subwindows. Scrollbars are windows (because they are visible) but they are usually thought of as properties of subwindows. Scrollbars do not manage the objects to which they are attached, it is the programmer's responsibility to make the screen updates associated with scrollbar actions.

An important feature of XView that is not a window is the *Panel*. Panels implement the OPENLOOK *control area*. Panels manage *panel items*, e.g. buttons, sliders, text fields, and other forms of inputting data. The motivation for panels is to provide a mechanism for propagating events, especially for objects which do not have a window associated them. Panels are very important in XView. For example, an application frame with no subwindows attached cannot catch interior mouse events. Attaching a panel to the frame allows these interior events to be propagated.

Obviously, there is much more to XView than what has been presented here, but this is sufficient to give the reader the necessary background for implementation.

4.4.3 Implementation Decisions. This major implementation decision encountered was to determine *how to make a way that the client-programmers can develop and add their new NN paradigms or algorithms to the ANNS system.*

4.4.3.1 Making a manageable development environment.

- **UNIX Socket:**

  UNIX sockets [43] provides the InterProcess Communication (IPC) facilities for independent executable programs. This idea was used to control NN paradigm processes for the ANNS system. The UNIX IPC interface makes IPC similar to file I/O. Each NN paradigm process has a set of I/O descriptors for reading and writing. The descriptor may refer to files, devices, or communications channels. So, this solution creates channels between the ANNS main processing

procedure and every NN paradigm component (subsystem). It provides for parallel development of simulation subsystem and is simple to implement. In this case, the NN paradigm subsystems are essentially stand-alone except for the IPC hooks to the ANNS main process.

- **The Interesting Events (IEs):**

  Given that NN paradigms are implemented as separate processes with IPC hooks to the ANNS main process, here comes up with a question: *What does a NN paradigm process communicate with the ANNS main process ?* To get a solution, the NN paradigm component needs some mechanism for reporting the Interesting Events for each process. In each NN paradigm subsystem, the BackGround main (BG_main(), see figure 30) procedure identifies and creates the IEs to communicate the ANN main process and update the simulation status to display graphical view on the graphical view window for every process step. Using multilayer perceptron paradigm as a example, the IEs would be INITIALIZE_NET for the first process step. The following processes would be MAKE_INPUT to the neural net, PROPAGATE the input vector, TRAIN_NET, and then DISPLAY_NETVIEW. The IE communication model that the ANNS main process communicate with each NN paradigm component is shown in figure 31.

- **UNIX Makefile and Archive Utilities:**

  After the client-programmer interface was created, another question is: *how to make a library manager and environment manager as in figure 8 of chapter III* ?Figure 8 shows an idealistic abstract model of the ANNS system in which a *library manager* provides an interface through which client-programmers manage NN paradigm components, and an *environment manager* provides an interface through which end-users select paradigm components to create interesting NN paradigm simulation. In software, the concept of a component library is extremely complicated and difficult to implement, because the new

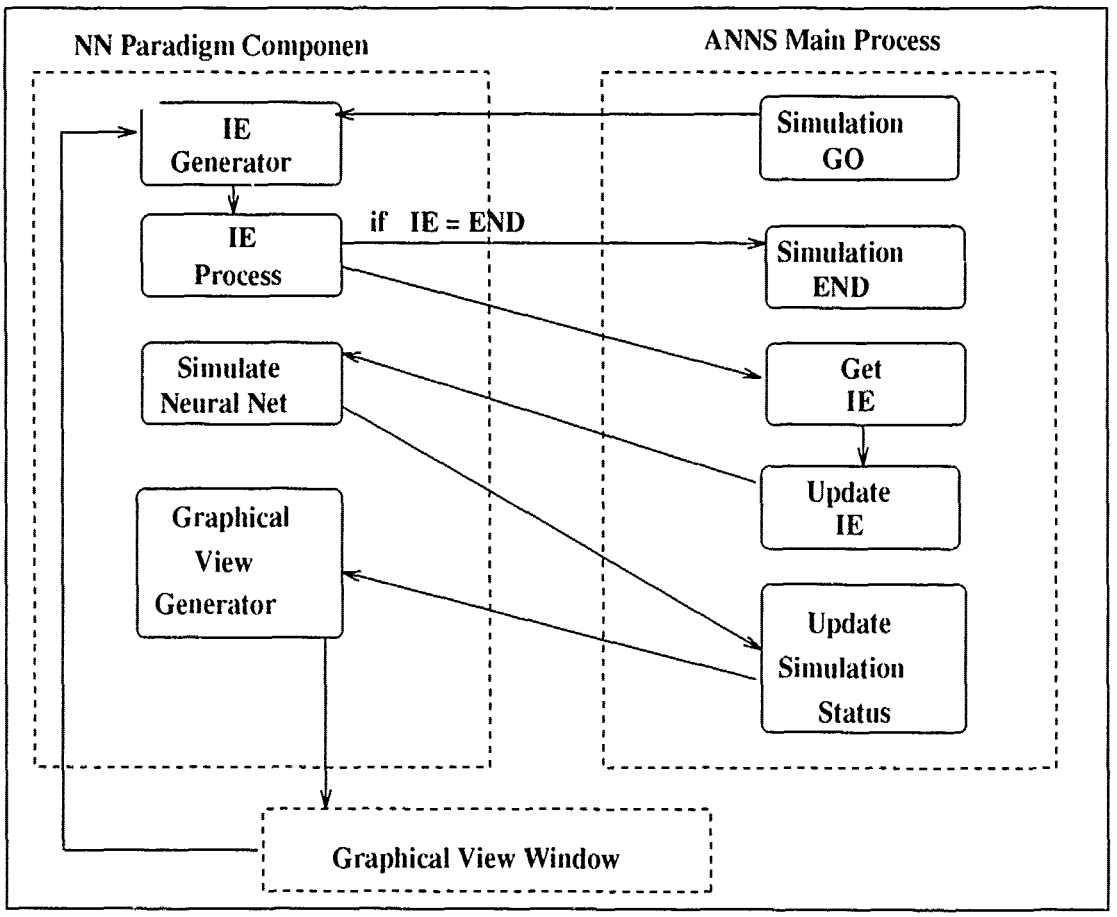


Figure 31. The IEs Communication Model

NN paradigm components would dynamically be added to the ANNS system by clinet-programmers. The solution for this case is to simply use the UNIX *Makefile* utility and **ar** (archive) utility. To do this, the ANNS main process, the common files library, and every NN paradigm component are organized as separate compilation units. This structure makes the ANNS system more modular, supports information hiding, makes files smaller and therefore easer to edit, and makes it possible to recompile only those compile modules that have been changed rather than the entire system. A portion of the top level *Makefile* for ANNS is:

```
BASE = $(PWD)
LIBS = ANNSmain ANNScommon
PARADS = BackProp Hopfield Hybrid Kohonen STP
all:
for d in $(LIBS) $(APPS);\
   do (cd $$d;\
    echo; echo  '****' compiling the $$d directory '****';\
    $(MAKE) BASE=$(BASE))\
echo;
        done
```

The ANNSmain and ANNScommon are library directories. The PARADS includes all paradigm components in the ANNS system. Each of these paradigm components is a dircetory with it's own *Makefile*. Another UNIX facility is **ar** utility, an important tool for promoting function level reuse for the C language. **ar** was used to bundle the ANNS main process object files into a library file with a single name called **libANNS.a**.

4.4.4 GUI Replacement Strategy from Silicon Graphics for Multilayer Perceptron Paradigm. Without a doubt, the overriding difficulty in this investigation was in understanding the source codes of *NeuralGraphics* and graphics routines of the Silicon Graphics System [39] [10]. There is precious little detailed end-users manual for *NeuralGraphics* , but there is not any documentation for client-programmers.

Weeks were spent pouring over code and doing execution traces using diagnostic print statements. If there were an anotated, graphical representation of the major data structure and control structure of the *NeuralGraphics* , the programmers could seriously benefit from understanding the flow chart of the system structure. Because the goal of the replacement process is to maintain *NeuralGraphics* in a orginally functional state, **no changes** to the NN algorithm structures of *NeuralGraphics* were attempted. In this case, the algorithm source codes remain completely functional. In addition to the GUI replacement, Parts of the efforts were in making connection to the ANNS interface for Interesting Events (IE) of the Net_Loop in the Backpropgation subsystem.

4.5 Testing Appraoches

This section presents the testing procedures. "Software tesing is defined as the execution of a program to find its faults" [12:191]. There are seven types of tests that can be performed on a software system [12:192-204]. The following discuss the tests performed on the ANNS system based on these seven types of tests.

4.5.1 Unit Tests. Also known as white box testing because the test is based on knowledge of the internal design of the module. According to the ANNS design methodology as in figure 7 of chapter III, unit testing was performed on the individual module to ensure the proper functionality was achieved. This included all the operations performed by the main menu buttoms and the operations performed by master control buttons.

4.5.2 External Function Tests. Validate the external system functions. This is also known as black box testing because the test performed has no knowledge of the internal design of the modules being tested. This type of test was used in conjection with the integration test described below.

4.5.3 Integration Tests. Validate the interfaces between system parts (modules, components, subsystems). It can be performed in one of three ways:

- **Bottom-Up**

  Each module is tested separately using special development drivers that provide the needed system functions. As more modules are added to the system, the driver is replaced by the modules that perform the simulated functions.

- **Top-Down**

  Uses a prototyping approach. A basic system skeleton is constructed and new modules are added and tested as they are developed. The function of lower level modules are simulated by program stubs.

- **Big-Bang**

  Each module is developed first. Then, they are all assembled and run together. This is the least effective of the three methods. The need for special drivers or modules stuba is eliminated: but, each module is only given a cursory test with this method and the likelihood of a totl system integration failure is great.

Integration testing for ANNS was performed using the top-down appraoch. The main subprogram was developed and implemented with stubs for each major function. As each module was developed. it was attached to the main system and tested to ensure proper integration.

4.5.4 System Tests. Validate the system to its initial objectives. The robustness of the system as a whole is considered during these tests. They take into consideration factors like peak loads and volume the system can accept, security, performace under peak and normal conditions, system reliability, error handling, and recovery mechanisms. The ANNS code was design to be internally robust. It tries to take into consideration all posible input events to the system. When a failure is detected anyway, a secondary mechanism takes control where the program

terminates as gracefully as possible. So, the system testing for ANNS was actually embedded into the unit and integration testing cases.

4.5.5 Acceptance Tests. Validate the system or program to the user's requirements. This test was done in conjunction with the installation test described below.

4.5.6 Installation Tests. Validate the instability and operability of the user's system. In other words, test the sytem in a real user's environment. This test was performed by getting several volunteers to use the system and fill out questionnaires pertaining their evaluation of the system based on a set of criteria. Appendix C contains a sample of the standard form used by the Department of Electrical and Computer Engineering at AFIT to evaluate software systems.

4.5.7 Regression Tests. Run a subset of previously executed integration and function tests to ensure that program changed have not degraded the system. Any time a major change or modification is made to a piece of module, there is the possibility of introducing errors into previously correct code. Thus, regression testing was made at each iteration of changes during the implementation stage. After the last iteration was completed, the ANNS system and the multilayer perceptron neural network subsystem were intergrated, and then the intergrated ANNS system testing was performed.

4.6 Results of Implementaion

This section presents the visible results of implementations based on the detailed desgins. All figures are directly derived from the actual screen images layout except the size of the images. See firgure 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, and 42.

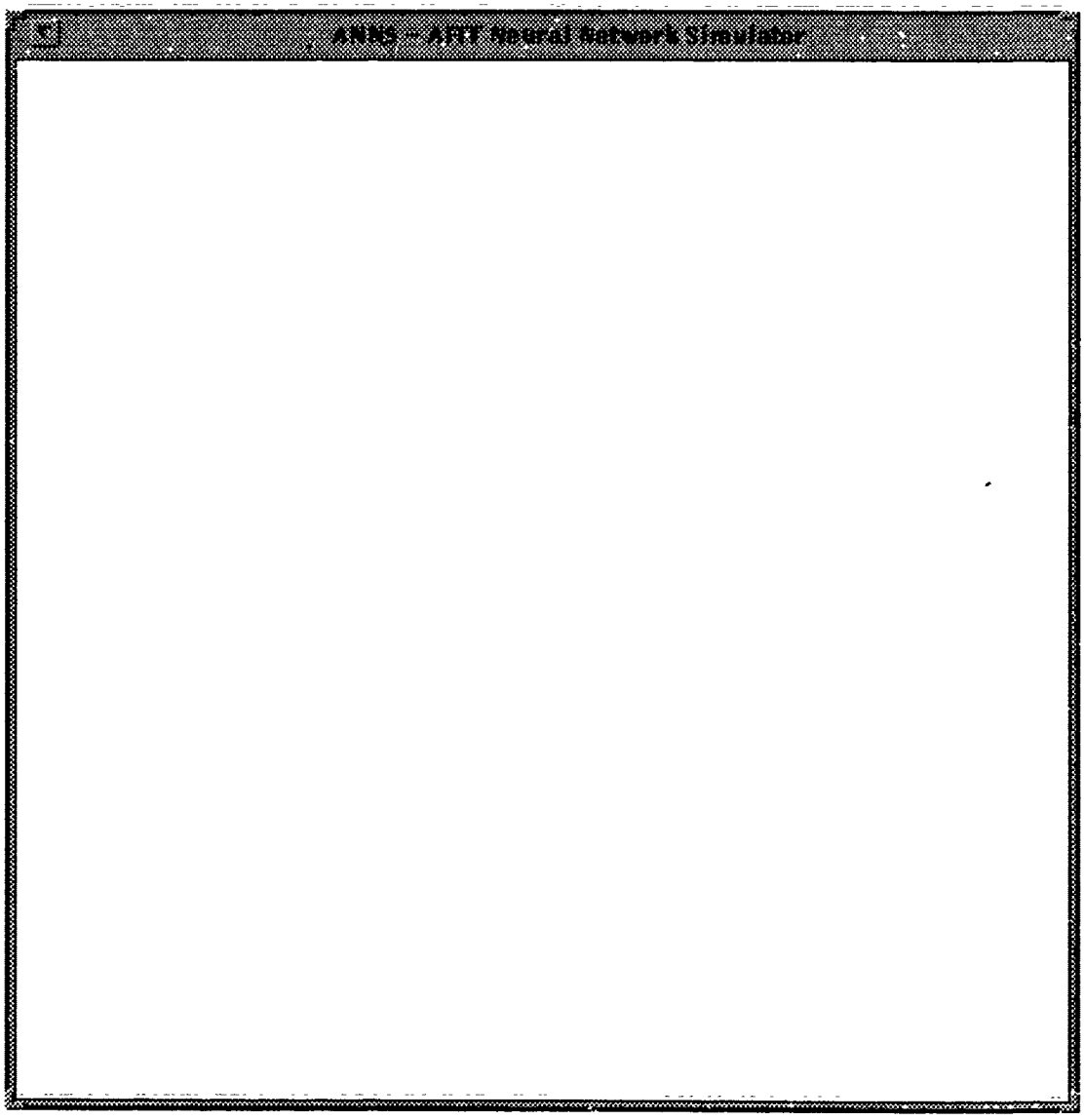


Figure 32. Main Window Model

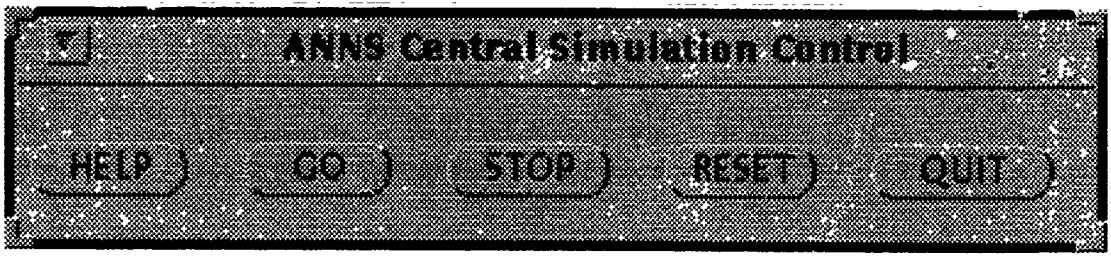


Figure 33. The ANNS Central Control Panel Model

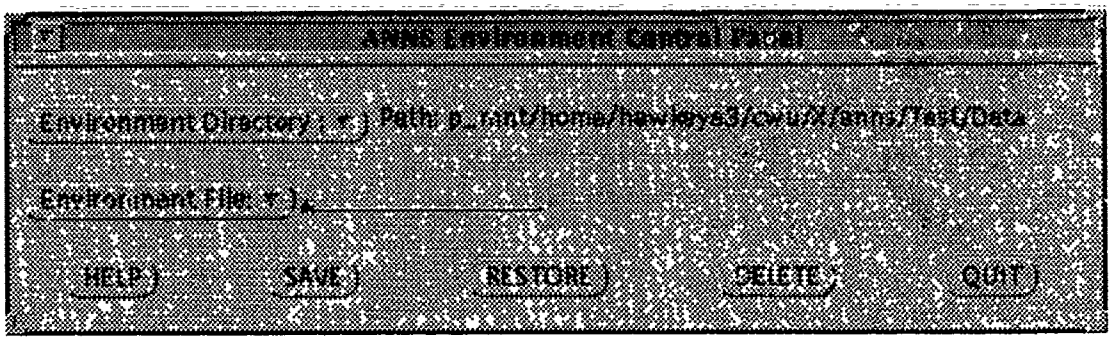


Figure 34. The ANNS Environment Control Panel Model

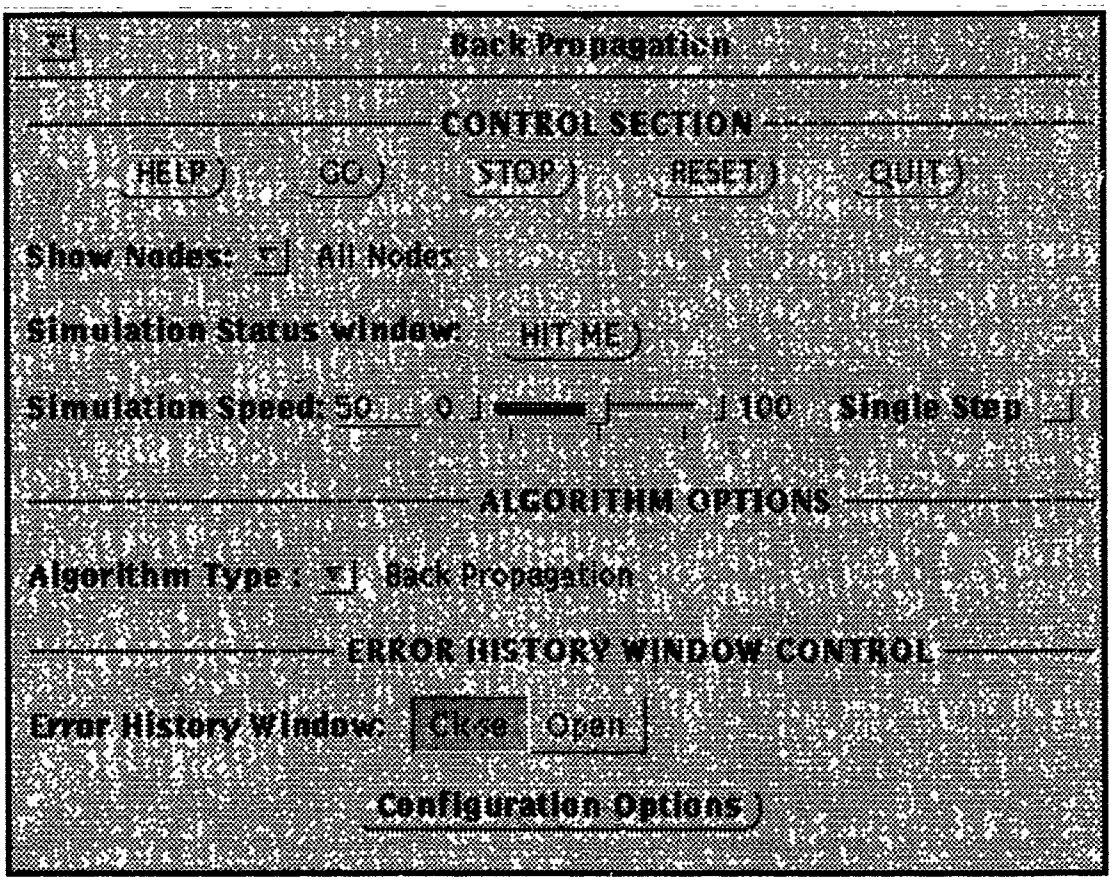


Figure 35. The ANNS Master Control Panel Model

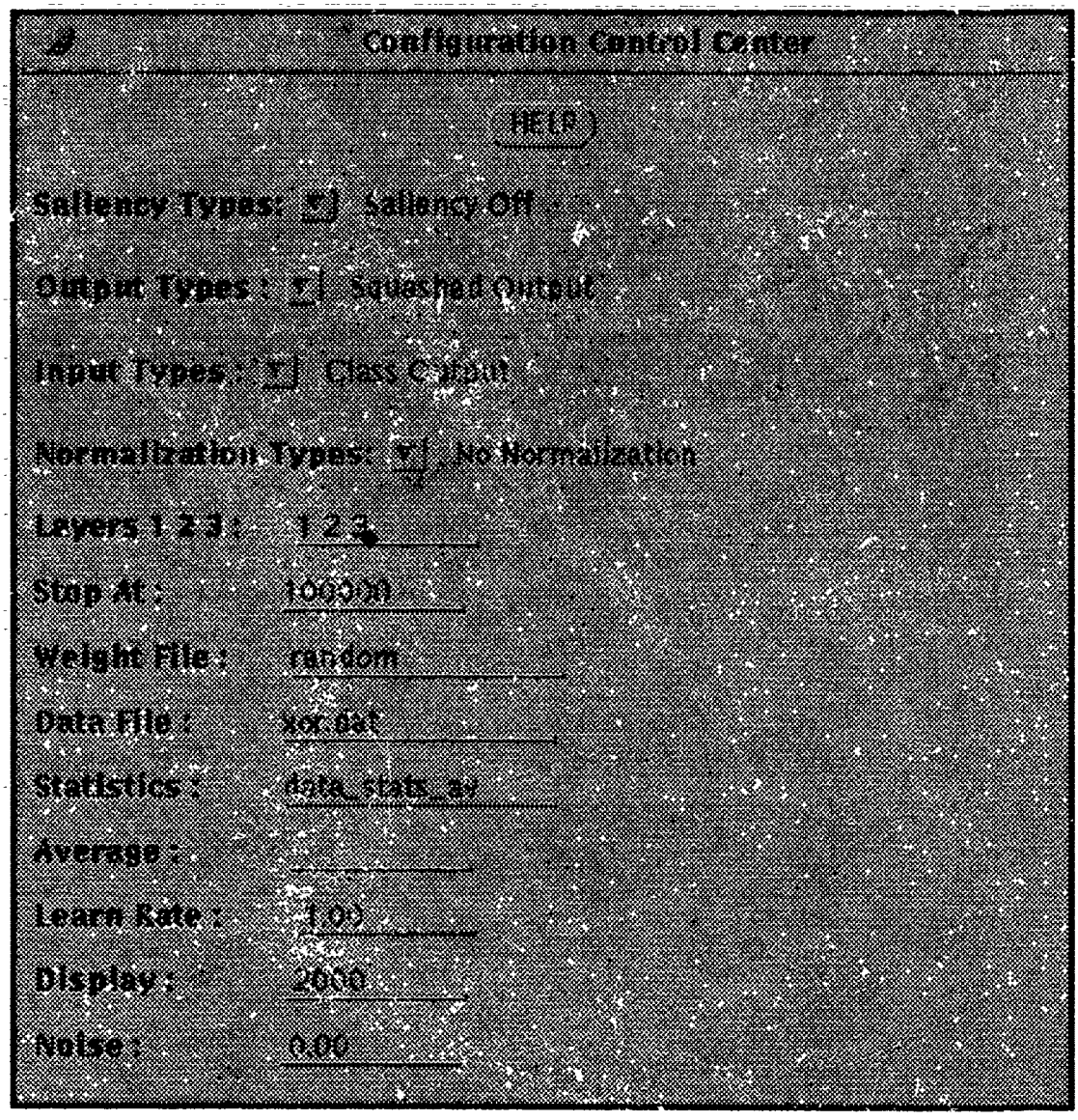


Figure 36. The ANNS Configuration Options Panel Model

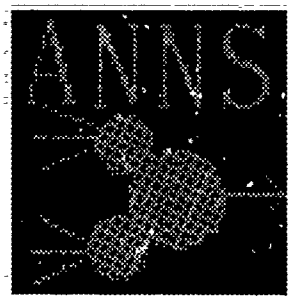


Figure 37. The ANNS Icon

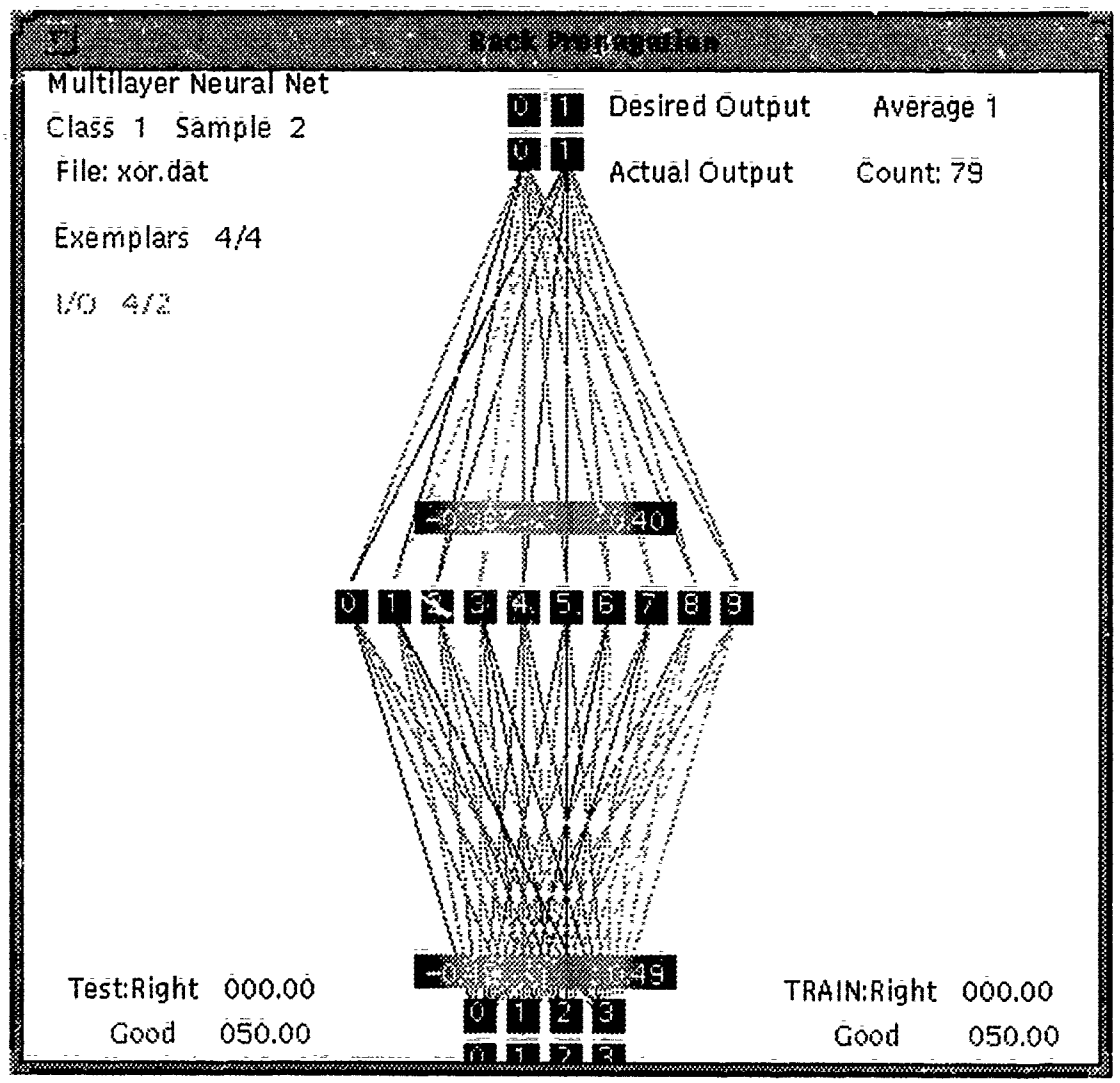


Figure 38. Algorithm Window Model

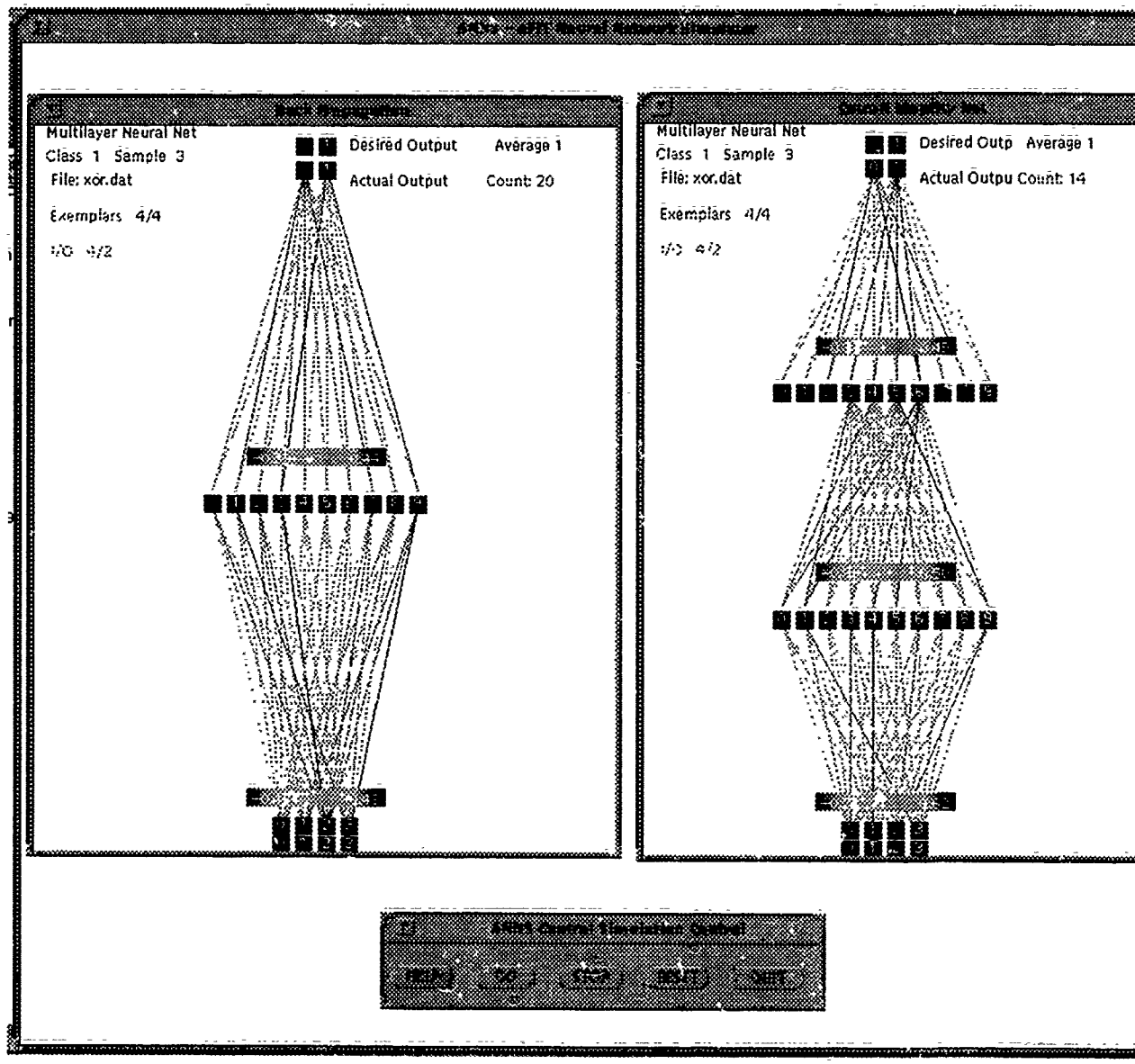


Figure 39. Two Algorithm Windows Comparison Model

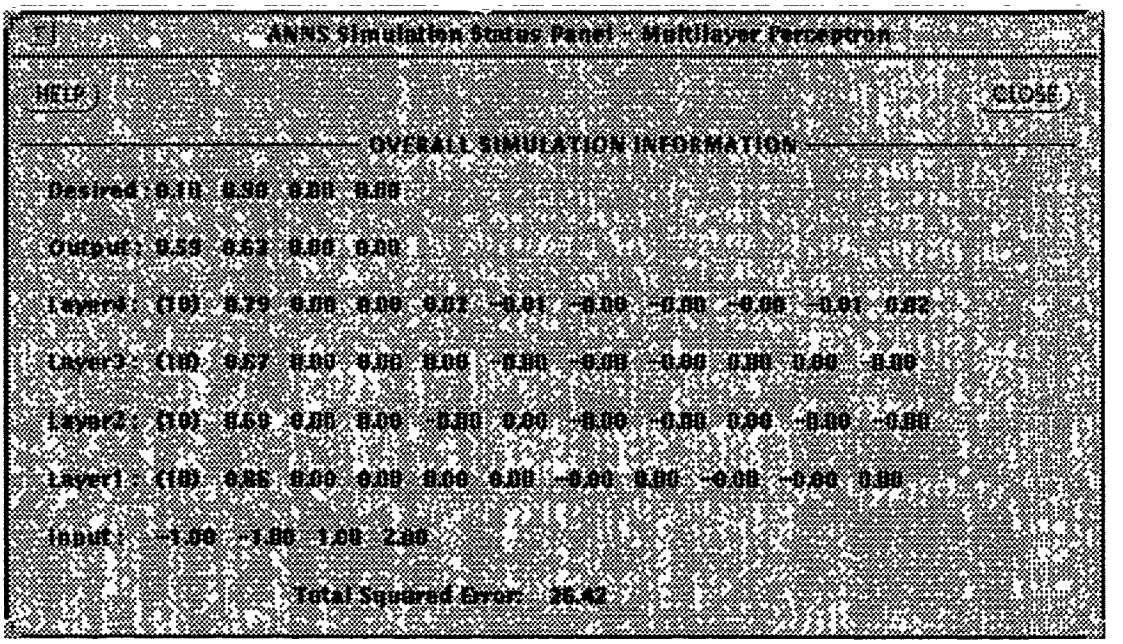


Figure 40. Simulation Status Window Model

4.7 Summary

This chapter presented the ANNS detailed object-orietned design, the motivations for selecting XView as GUI. and implementation. Implementation essentially consisted of the top-down mapping of ANNS design objects to their corresponding XView objects. The dependencies of the design modules were also represented by graphical figures.

The results of the implementation presented the actual screen images layout for mapping the detailed object-oriented designs. The implementation decisions and the test procedures were also discussed. In the next chapter the research summary of the ANNS system is presented, and recommendation for the further research are offered.

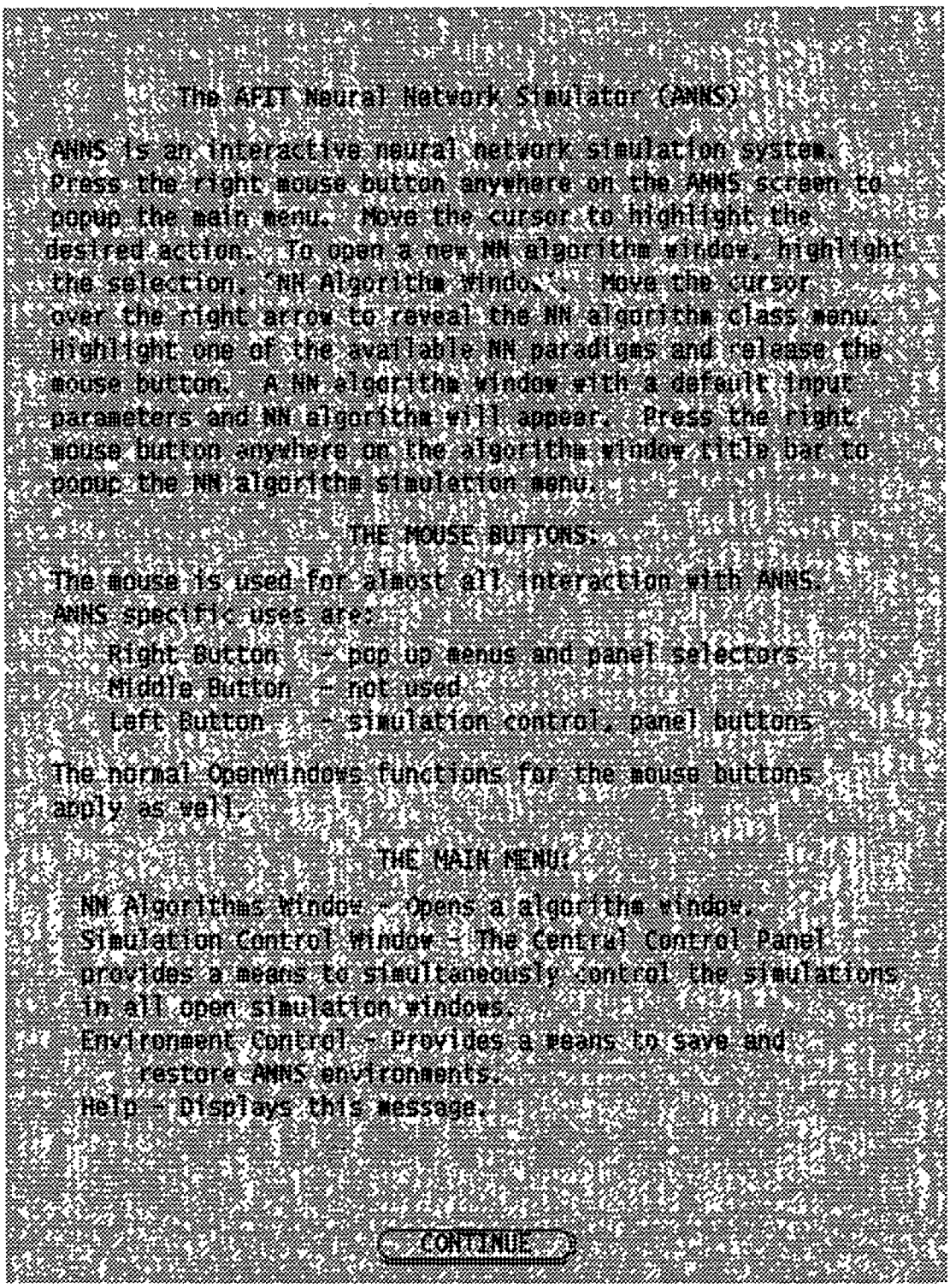


Figure 41. On-line Help Window Model

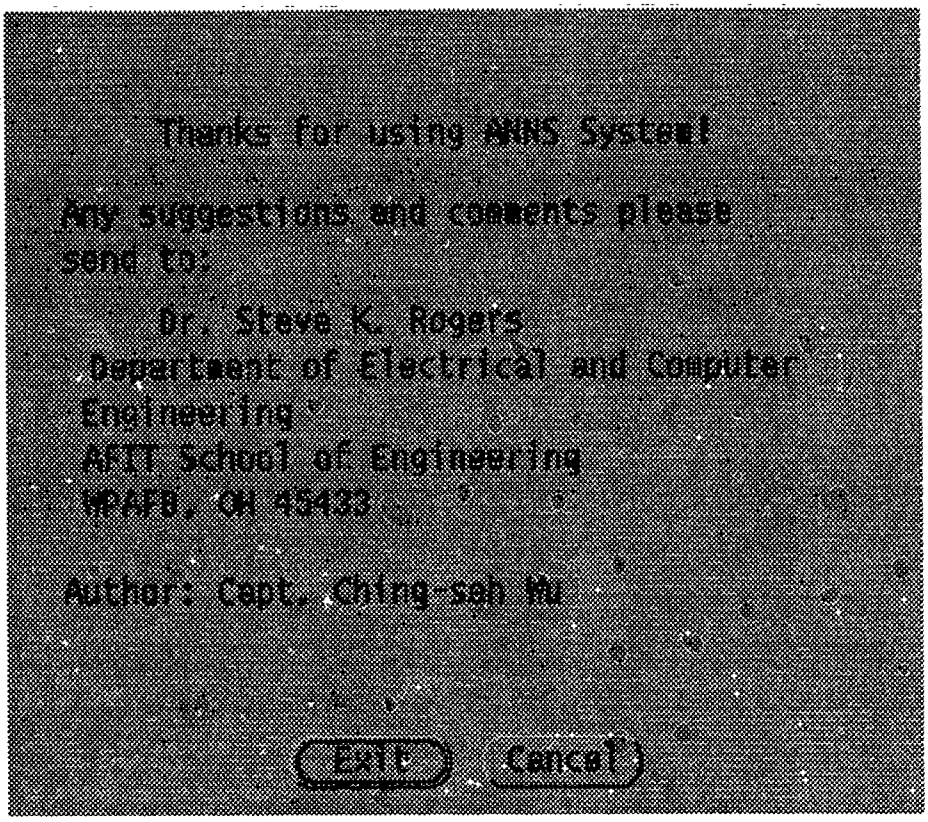


Figure 42. Exit Notification Window Model

V. Conclusions and Recommendations

5.1 Introduction

This chapter summarizes the work performed for this thesis. It then presents some recommendations for further research.

5.2 Research Summary

This thesis effort resulted in the development of an animated graphical user interface for the neural network simulator. It provides engineers to the neural network field of graphical demonstration to illustrate some of the concepts in neural network problem solving. The results of the ANNS implementation have met or exceeded the requirements established in Chapter III. The work to generate this project was accomplished using an iterative approach. The major actions performed were:

- *Analysis of the problem domain.*

  Before doing any design or implementation, an intensive study was accomplished to gain an understanding of the terms, concepts, and philosophies needed to design a user interface for ANNS. This involved research in the areas of neural networks, graphical user interface design, object-oriented design, X Window System, the *NeuralGraphics* package.

- *Determination of system requirements.*

  The system requirements were gathered through the series of literature reviews of simulator systems, and based on the idea to create an integrated and unified neural network simulator for both end-users and client-programmers.

- *Development of the high-level design.*

  Having laid the foundation through the accomplishment for the previous two steps, the next task was the high-level, object-oriented design of the ANNS system. This involved the identification of the objects, their attributes, and

the communication needed between them. At this point, several alternatives of how to incorporate the X Window System into the design were considered and the overview of the ANNS system was determined.

- *Detailed object-oriented design.*

  Before the detailed design, the motivations for selecting XView as graphical user interface was discussed. This discussion based on the users perspective and programmers perspective. For the detailed design stage, the *module diagram* was used to illustrate the allocation and dependency of the classes and objects.

- *Implementation and testing.*

  The implementation stage was to create and map objects from the detailed object-oriented design, and then generate the software source codes. At this stage, the types of objects of XView associated with this detailed design were discussed, some of the technical implementation decisions were made and the GUI replacement strategy was depicted. This overall system was generated through a repetitive process of evaluation and planning, detailed design, code generation, and unit, regression and integration testing. This method ensured that a working product was available at the end of each iteration. At the last step, the visible results of ANNS modules were presented.

5.3 Recommendations for Further Research

This development of effort provided interfaces that the end-users can simulate this system friendly and the client-programmers can maintain the ANNS system or add new neural network paradigms into ANNS easily. While this ideal design of the ANNS system is a good beginning, there were certain features and neural network paradigms that could not be implemented due to time constraints. Thus, the following ideas are provided as areas of potential research.

5.3.1 Develop and Integrate all NN Paradigm Components into ANNS. ANNS is too big, the time at AFIT is too short, and the learning curve is too steep to attempt major design changes in concert with the GUI replacement for all the NN paradigms running on the personal computer version during this thesis cycle effort. Therefore, these remained jobs would be finished by the client-programmers or next thesis cycles.

5.3.2 A Network Version of the ANNS. One major benefit of using X window system is that it has built-in support for distributing the NN processing and display activities over several systems connected by a network; one X client can connect to multiple X display servers. In words, users can process the ANNS system from different remote resources by computer networks if there is an existent network version of ANNS.

5.3.3 Portability Considerations. Since C++ programming language has become more popular and supported as a programming environment, changing all graphical modules of the ANNS system to C++ will enhance the portability.

5.4 Conclusions

In conclusion, this thesis documented the object-oriented design and implementation of the ANNS system. This work forms the baseline for future efforts at completing an integrated AFIT Neural Network Simulator (ANNS) that can help teach neural network theories (from end-users point of view) and help research the different kinds of neural network paradigms (from client-programmers point of view).

The ANNS system is a multi-application tool which allows rapid study of segmentation, vector quantization, and other Neural Network paradigms. According to the literature review, there are only very few versions of the neural network simulator running on Sun workstations nowadays. As many organizations are purchasing commercial software packages to perform these types of applications, making ANNS

system with all kinds of NN paradigm subsystems available could save the U.S. Air Force thousands of dollars.

Appendix A. ANNS User's Manual

A.1 Introduction

The AFIT Neural Network Simulator (ANNS) is a development and tutorial system for the study and research of artificial neural networks. It is also an integrated and graphically interactive neural network paradigm simulation system. ANNS allows the end-users to select the optional neural network paradigms and algorithms for simulation, and allows the client-programmers to add new neural network paradigms and algorithms to ANNS. **This manual is primary for end-users;** it only describes how to use ANNS. Client-programmers should also refer to the ANNS Programmer's Guide for the overall ANNS implementation and general procedure for adding a new neural network paradigm to ANNS.

The ANNS system is intended to test feature extraction methods, compare training paradigms, and help the user understand the limitations and utility of this novel approach to computing. ANNS runs on Sun SPARCstastions using the SunOS (Sun Microsystem's version of the AT&T UNIX operating system) and *OpenWindow* environment at Air Force Institute of Technology (AFIT). There are several control mechanisms provided. including GO simulation, STOP simulation, RESET all parameters, variable simulation SPEED, single simulation STEP, and QUIT the simulation. Other features include:

- a NN paradigm class selection from a list of available classes stored in the paradigm library directory, such as *back-propagation, hybrid training, radial basis function, hopfield associative memory, etc.*.
- Dynamical simulations with color monitors.
- Multiple NN simulation windows, each of which with one master control panel.
- Master control panel for modifying the input parameters and control parameters, and for monitoring the simulation status associated with each NN paradigm.

- Simulation environment save and restore capability.
- Simultaneous comparison and control of multiple Neural Network (NN) paradigm and algorithm simulations.
- Provide on-line help for every interface function and window.
- Provide error window to illustrate the error surface.
- provide simulation status window to show the static simulation status at each training step.

A.2 Backgrounds Needed for Users

This section presents some backgrounds related to the paradigms in the ANNS system.

A.2.1 Ideas of Computer Gambling. The idea of using a computer to predict the outcome of football games, horse races, or the stock market has fascinated computer enthusiast since the beginning of the digital age. Before computers, statisticians tries to relate measures of performance to probabilities of outcome using multivariant linear regressions and Bayesian analysis. But does it work ?

A.2.1.1 Can computers predict winners ? The answer should lie in your own common sense. One thing computers can do very well is calculate probabilities of future events based on a record of past events. Before neural networks, those calculation were complicated to say the least. But with neural network, algorithms for static pattern recognition, and function approximation, nice implementation of prediction systems are possible. More importantly, the user doesn't have to know the nitty-gritty detail to make things work.

Prediction is possible because neural networks relate inputs to output. With a history of events, measurement of performance and a history of outcomes, it is a simple matter to train a net to predict similar events. The prediction may be wrong,

but it can from the probability perspective, the most likely outcome. The trick is to use the right inputs. Even the magic of neural nets cannot overcome the computer age axiom, garbage in-garbage out.

A.2.1.2 Using the Artificial Neural Networks. Several type of prediction problems have used neural network solutions. This involves what is called static pattern recognition to predict probabilities and function approximation to predict the behavior of indicators like the Dow-Jones Industrial Averages.

Using a neural network to compute these probabilities (and functions) is based on two fortunate properties of the network. First, the output is base on a sigmoid function which is a member of a class of functions know as conditional density functions. The integral of the sigmoid function is a probability density function, the work horse of probability calculations.

The second property is that the neural network can be treated as a black box. The number crunching is always the same. The network training and propagation rules are built into the computer program. All that is required of the user is to prepare the input data, and turn the net loose. When the network has trained enough to correctly identify the training data, simply propagate some unknown data though the network.

Sound easy ? Well it is. Only one problem remains, picking the right input data. A few sample problems might be in order here, but first, you need a little background.

A.2.1.3 Pattern Recognition-A Three Headed Beast. Pattern recognition is three part problem, segmentation, feature selection and classification. Segmentation is to remove the event from the background. It's pretty easy for most prediction problems, just read the paper. Event descriptions are right there. The same is true for the feature extraction phase. Feature extraction is the process of

taking a meaningful measurement on the segmented event. Again, this is pretty easy, since the paper is full of measurements which describe what took place at the event. Words like batting average, and point spread come to mind.

Feature extraction has one small problem, the operative word is "meaningful". A measurement of sunspot activity near Mercury probably would not be related to the outcome of a horse race. It's easy for unrelated measurements to creep into the pattern.

Pattern Recognition is to take a set of measurements of an event (a pattern) together with a specified outcome, and predict the outcome for similar patterns whose outcome is not known.

For a baseball game the pattern could be a set of numbers like the past win/loss ratio, earned run average, team batting average and so on. The outcome would be, (given the above information) winning or losing.

If the event is frozen in time, that is, a simple set of measurement and an outcome the pattern is referred to as a static pattern.

Gambling can be performed in two manners: static pattern recognition and function approximation. The first is an application which picks a winner based on a set of measurements. The concept can be extended to include static pattern comparison. Pattern comparison would use the two sets of measurements, side by side, more or less, to predict a winner. Function approximation takes a set measurement and tries to estimate a actual number. The difference between the two might be considered as the difference between, predicting the winner of a football game and predicting the point spread. As you might guess, the second is quite a bit harder than the first.

A.2.1.4 Static Pattern Classification. This section will examine three types of problems, assignment of risk, generalized pattern classification, and comparative pattern classification.

Static pattern recognition works with a yes/no type of predictions. It would answer questions like "Is this a good stock to buy or not ?" "Will this team go to the superbowl or not ?" The generalization of this is standard classification problems. Given a set of measurement, assign each set to a particular class.

A.2.1.5 Function Approximation. Function approximation takes a pattern and tries to predict a future value of the function. Function approximation has the advantage that it can use itself as the pattern. For example if one wanted to predict the interest rates for next year, the input to the net could be time samples of last years behavior of the interest rate function. At the same time, it could be similar to a static pattern classification, except the output would be not be yes or no but an actual number.

Many have tried to use function approximation for prediction, based on past performance usually reporting poor results. Those reporting good result are generally trying to sell neural network models or acquire grant money.

Better results are usually obtained by including more information from past performance. One experiment predicted the stock market using a double input net [11]. On one set of inputs a time history was used. On the second set of inputs, market measurements were used, i.e. interest rates, GNP, and so on.

A.2.1.6 Example:Good Stock/Bad Stock. Procedure: Get a year old newspaper and select a few hundred of your favorite stocks. Take the measurements listed there, such as P/E ratio, cost, change or what ever you can get.

Now get current papers and classify each of the stocks that went up as class one and those that went down as class two. Now setup your data file as shown in the example data file in section A.4.1.

Allow the net to train until the best accuracy is obtained. Now use the test functions to enter measurements for this years data.

A.2.1.7 Example:Predicting a Baseball Game. One example tried with simulated data is to examine a number of baseball games[11]. First, an eight entry wide feature vector is made. The first four positions are four measurements of the skill of team 1 (which played team two). The measurements are win/loss ratio for the last ten games, team batting average, earned run average and season win/loss ration. Team 2's measurements went into the last four places of the feature vector. Those numbers were chosen because they were available in the newspaper. Since the outcome of the games are known, it can be used to train the network. Here is a case which can statistically normalize the data. On the pretend data the problems seemed trainable over a wide range of noise added to the input.

The real problem is that under the right conditions, any team can beat any other team regardless of the features used to train the net. So much of the data used to train the net will be contradictory.

When a network is trained with contradictory data, the output of the network is much like a probability density function. For example, if there is a data set like this (see data format example in section A.4.1):

```
2 0 2 2
1 1.0 1.0 0
2 1.0 1.0 1
```

The first training point says that the input pair (1.0,1.0) should map to an output of zero. The second data point says, that it should map to a one. The result is (usually) the output will go to 0.5 for both, which might be interpreted as a fifty percent probability of being either outcome.

A.2.2 Multilayer Perceptron Paradigm. The multilayer perceptron is a feedforward network based on the work of Paul Werbos, and working separately, David Rumelhart and James McClelland [19]. A Muitilayer Perceptron network, sometimes called backpropagation, takes a feature vector as input and tries to learn a correct classification by adjusting the interconnecting weights between layers of

independent processing units. By presenting feature vectors and desired outputs to the network, interconnection weights are adjusted based on an error term generated by the difference between what is desired and what the net actually produced. The computation elements for a neural network model are called perceptrons. NN comes in a variety of flavors. The most common NN is the backpropagation perceptron, but other common types include Kohonen nodes, Counterpropagation nodes and Radial Basis nodes. Some definitions refer to NN as always being backprop, here the term will be used in a more general sense.

A.3 Getting Started

A.3.1 Set Path. Set your UNIX path variable to include the ANNS executable directory. So far, there is only one executable ANNS in /home/hawkeye3/cwu/ANNS/bin directory. Copying the executable ANNS to your own directory is strongly not recommended, because it consists of several executable subsystems and comsumes a lot of memory space. Example of setting path:

```
set path=( $path /home/hawkeye3/cwu/ANNS/bin )
```

A.3.2 The Mouse. The mouse is used for almost all interaction with ANNS. ANNS specific uses are:

- **Right Button:** pop up menus and panel selectors
- **Middle Button:** not used
- **Left Button:** simulation control, panel buttons

The normal OpenWindows functions for the mouse buttons apply as well, such as *click* (push and immediately release) or *depress* (push and hold until some action is complete).

A.3.3 The Main Window. After setting the path, typing **ANNS** at the UNIX prompt and hit "Return" key will come up with the ANNS main process window. (See figure 43)

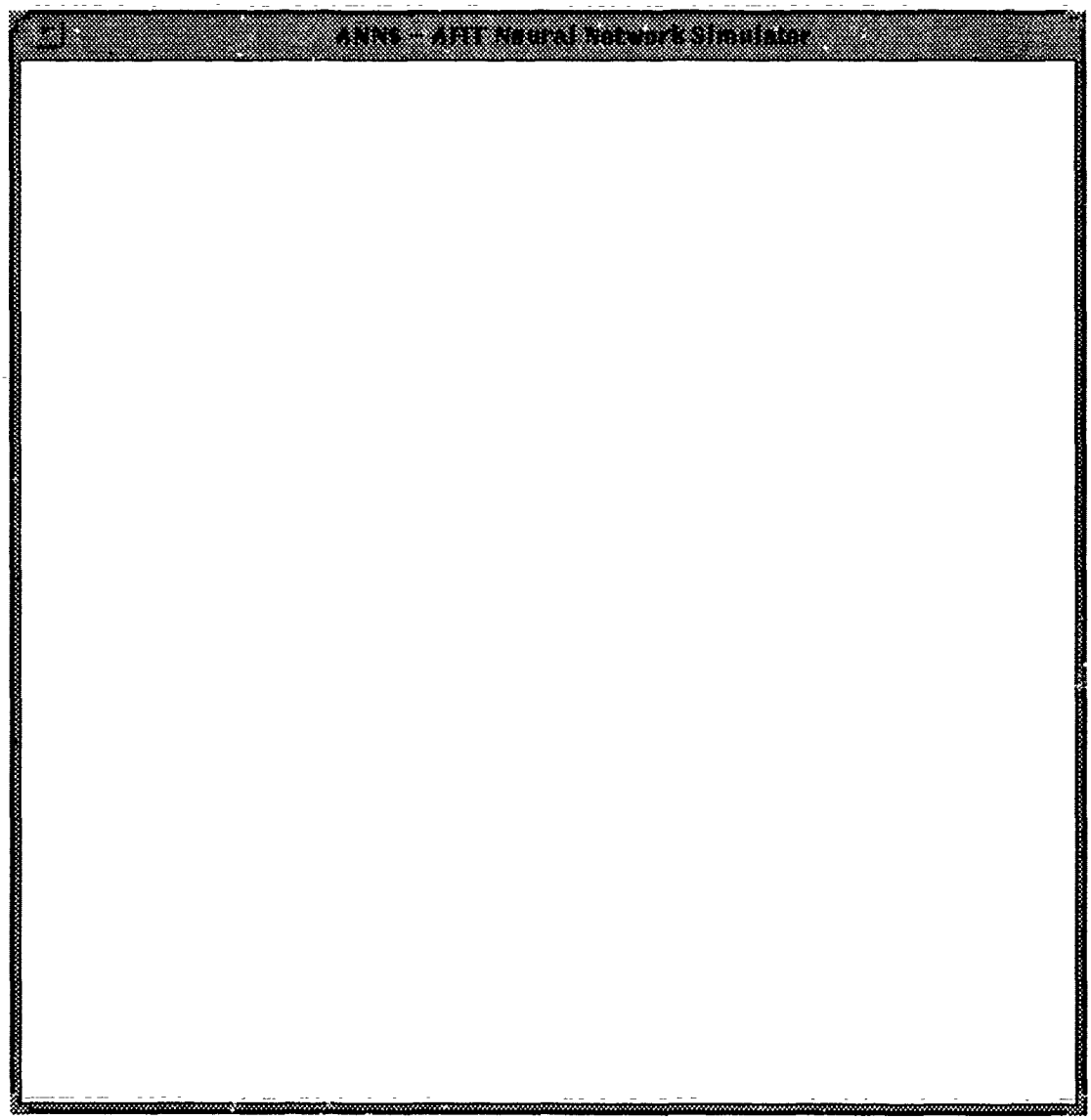


Figure 43. Main Window

A.3.4 Iconify ANNS. This action causes ANNS and all simulations to stop and become an icon. (See figure 44) The icon may be moved anywhere on the screen. ANNS is deiconified by clicking the right mouse button on the icon image.

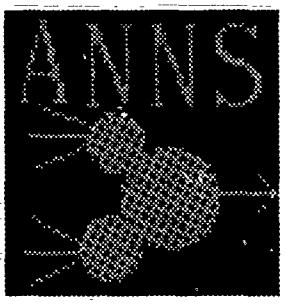


Figure 44. The ANNS Icon

A.3.5 The Main Menu. ANNS uses menus to allow users to make a selection from among several choices. Users *pop up* menus by depressing the right mouse button. Generally, menus remain visible only as long as the right button remains depressed. While a menu is visible and the right mouse button is depressed, moving the cursor over a particular menu entry causes that entry to be highlighted. Releasing the right mouse button with a menu item highlighted *selects* that menu item.

A menu entry with a right-arrow indicates that a *pull-right* menu is associated with that menu item. Moving the cursor over the right-arrow exposes the pull-right menu from which a selection can be made. Usually, the first item in a pull-right menu is the default selection for an item with a pull-right menu.

Press and hold the right mouse button with the mouse pointer anywhere within the ANNS main window to pop up the main menu. There are five choices in main menu:

- Neural Network (NN) Algorithms Window
- Simulation Central Control Window

- Environment Control Window
- Exit ANNS
- Help

A.3.5.1 Neural Network (NN) Algorithms Window. To open a new NN algorithm window, highlight the selection, "Neural Network (NN) Algorithms Window". Move the cursor over the right arrow to reveal the NN paradigm class menu. Highlight one of the available NN paradigms and release the mouse button. A NN algorithm window with a default input parameters and NN algorithm simulation window will appear on screen. The default input parameters are shown on the configuration window by clicking the "Configuration Options" panel on the algorithm window. If there is no change desired on the algorithm window (or master control panel, see figure 45 and 46), simply click the GO panel item or click anywhere on the simulation window, and then the simulation runs with graphical view display on simulation window. (See figure 47 (Be sure the setup data file is in the same directory as this simulation runs)

A.3.5.2 Simulation Central Control Window. Clicking this item, the Simulation Central Control Panel will appear. This panel provides a means to simultaneously control the simulations in all opened simulation windows. (See figure 48 and 49)

A.3.5.3 Environment Control Window. This selection causes the Environment Control Panel to appear. This panel provides a means for saving the current simulation environment, restoring a saved environment, or killing a saved environment. The environment includes all users options currently selected: window size, window placement, and NN algorithm parameters. The environment control panel is shown in figure 50.

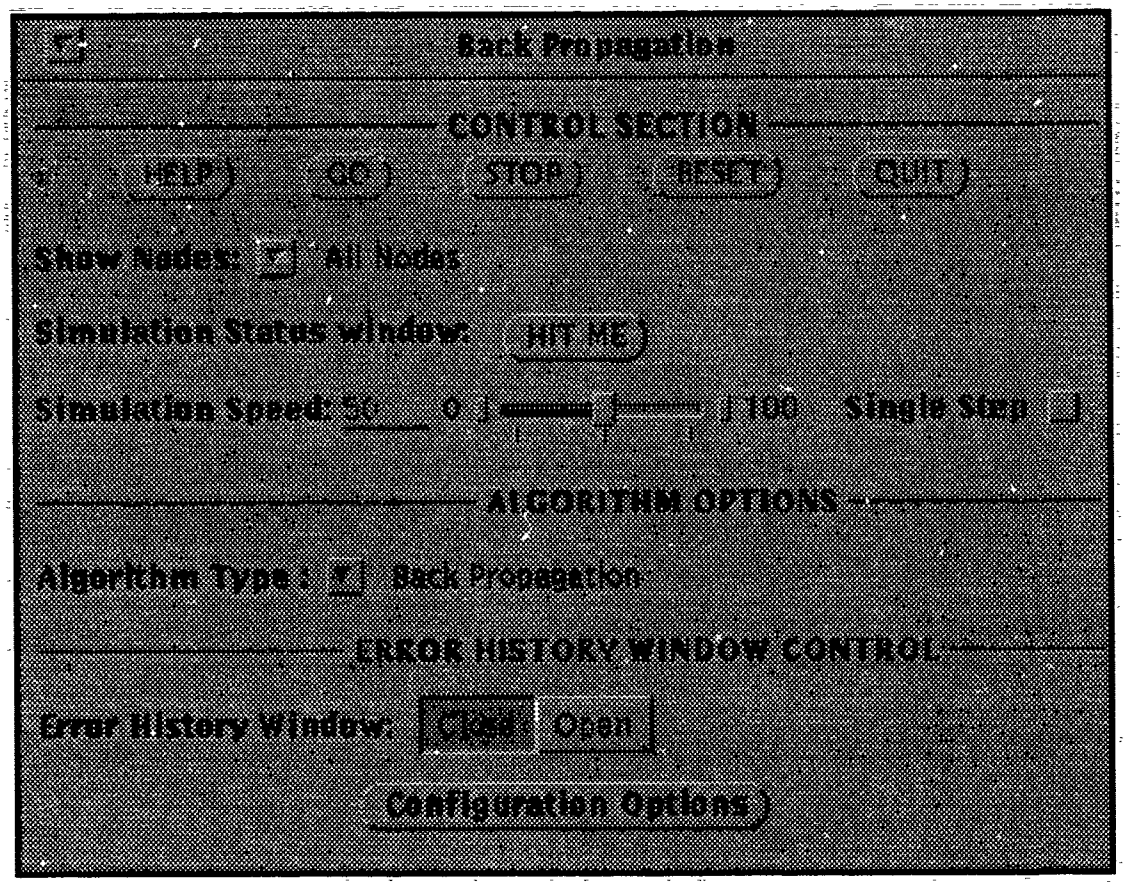


Figure 45. The ANNS Master Control Panel

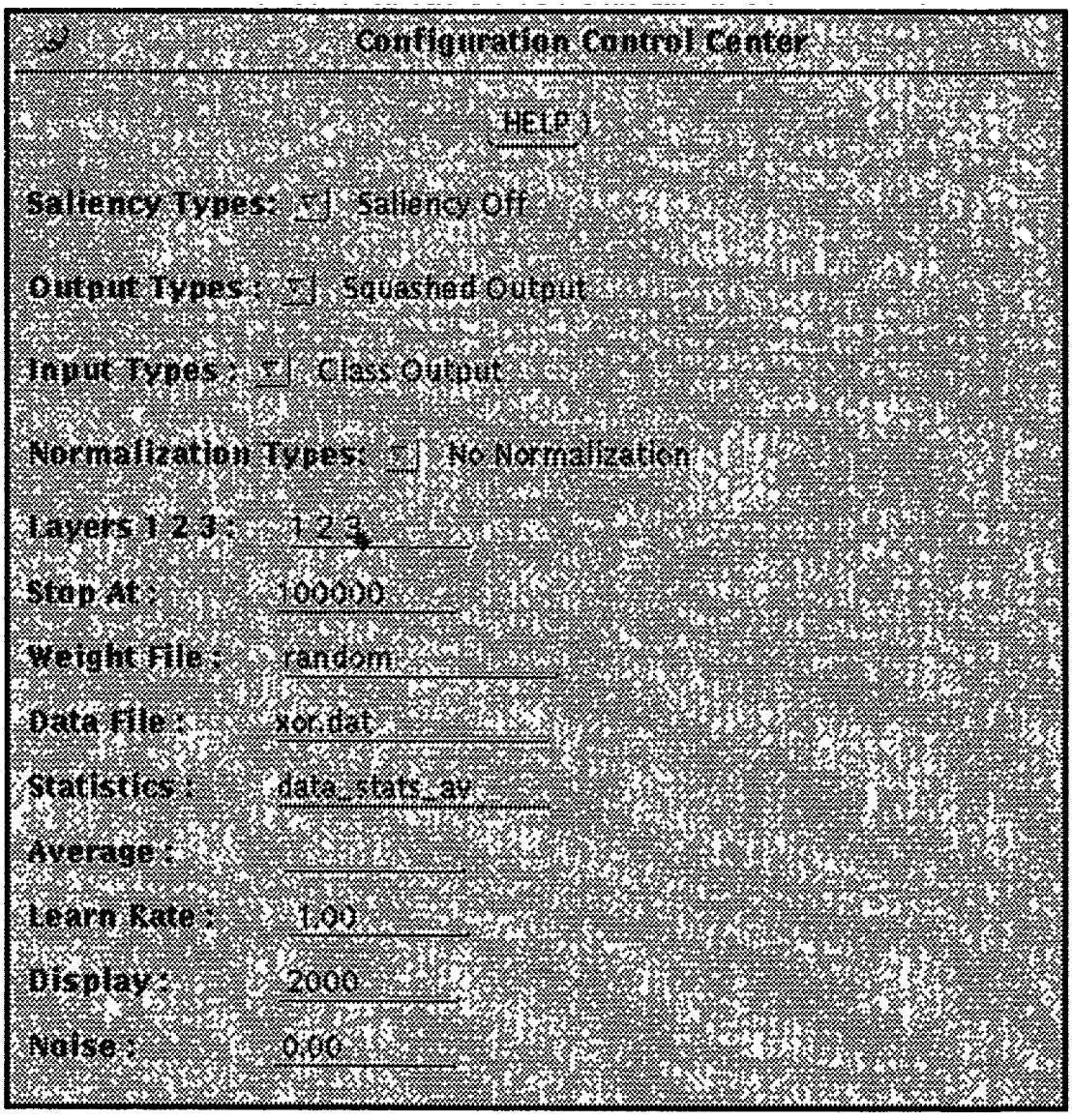


Figure 46. The ANNS Configuration Control Window

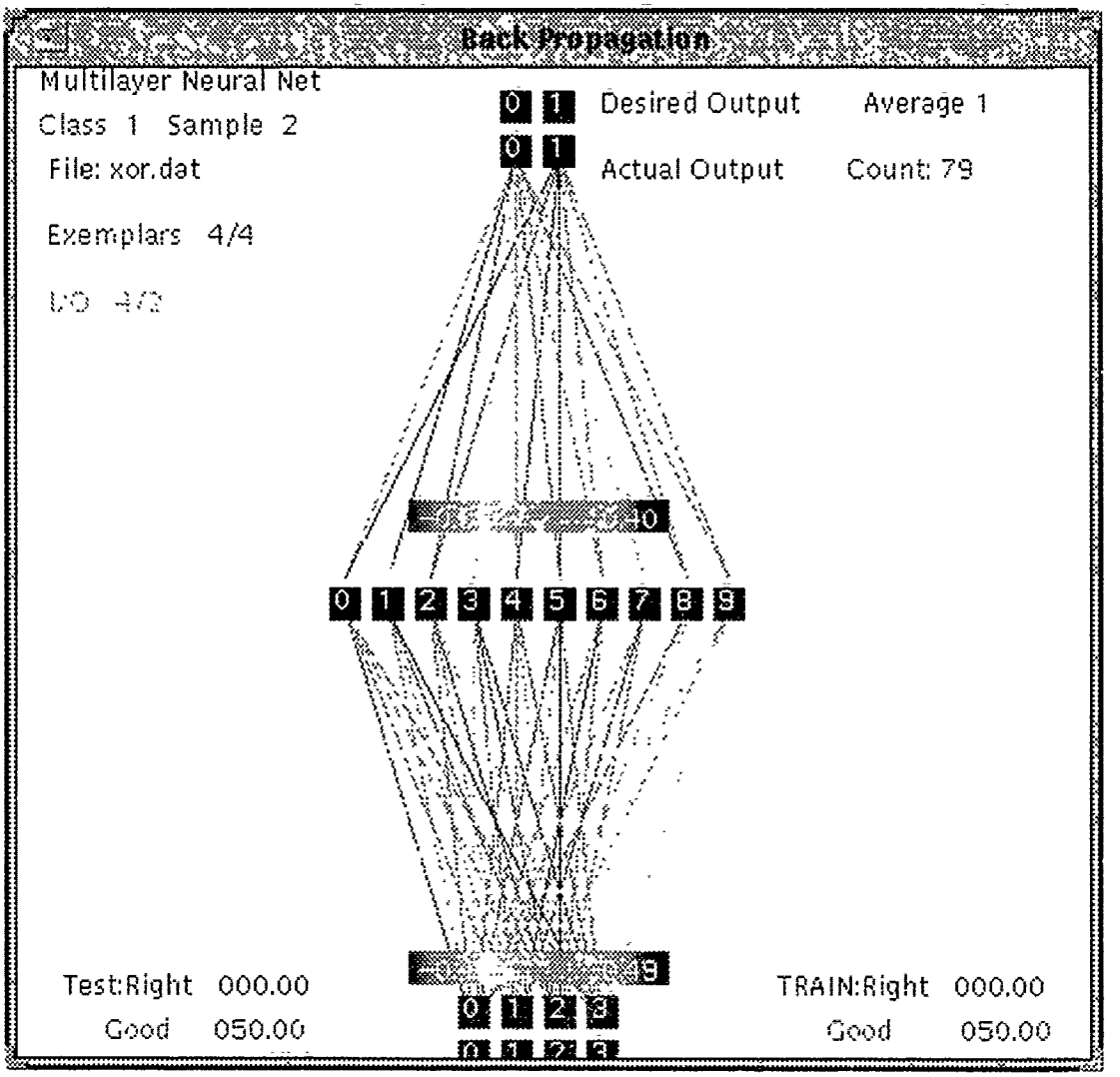


Figure 47. Simulation Window

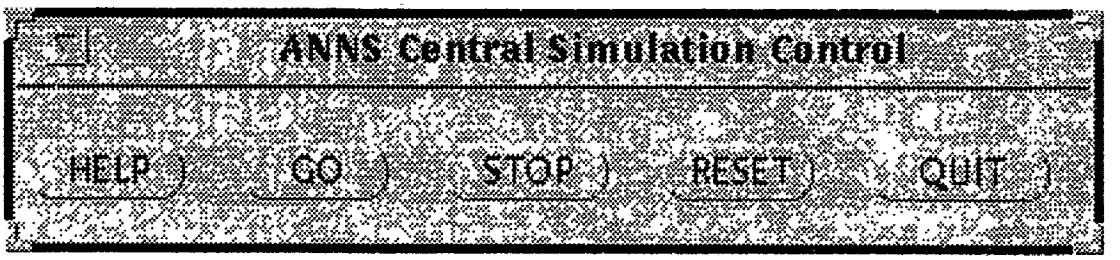


Figure 48. The ANNS Central Control Panel

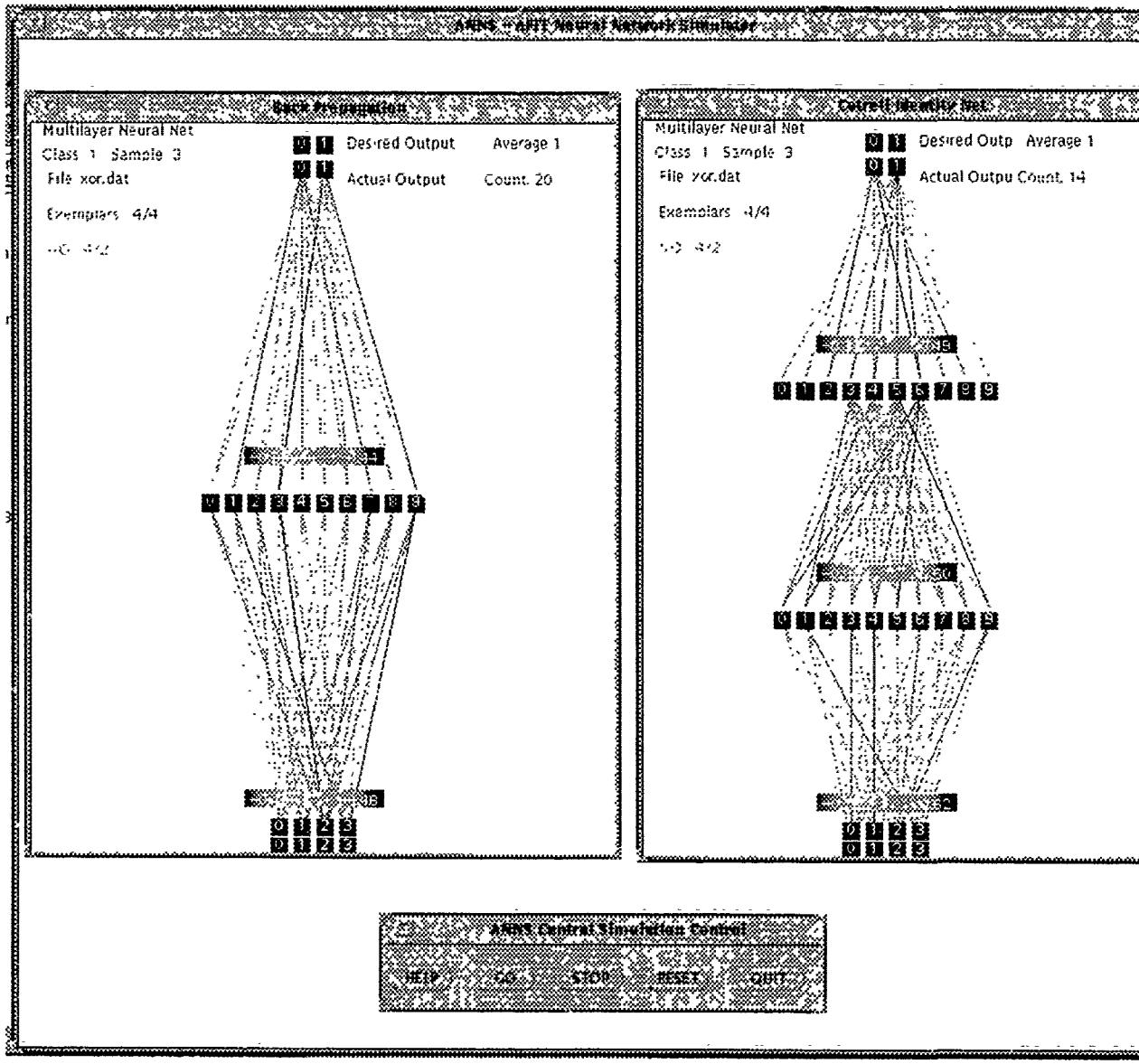


Figure 49. Two Simultaneously Simulation Windows

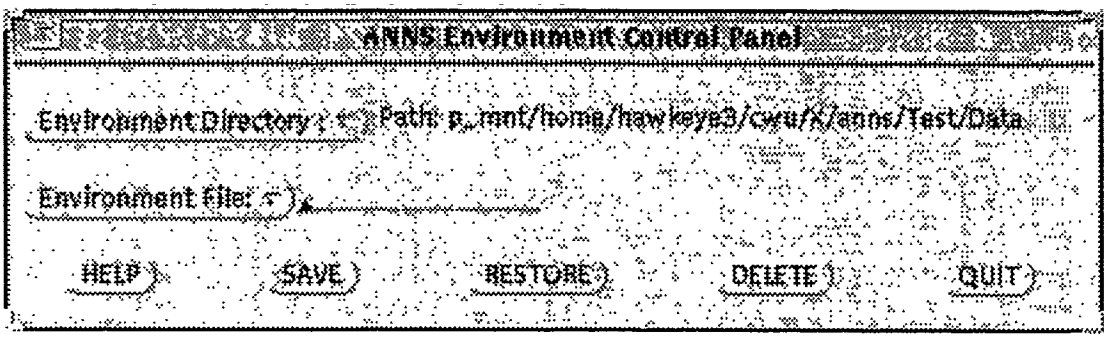


Figure 50. The ANNS Environment Control Panel

A.3.5.4 Exit ANNS. This item will end the simulations and exit ANNS. The user is first notified to confirm the exit. (See figure 51)

A.3.5.5 Help. This item displays a help window that briefly explains the mouse actions, the function of each main menu item, and how to open a NN algorithm window. (See figure 52)

A.3.6 Master Control Panel. Figure 45 shows the master control panel. This panel is divided into several section depending on the type of NN paradigm.

- **Control Section:**

  Provides several panel items for controlling the simulation. The HELP panel item is for on-line help briefing discussing how to use this master control panel and what the function is for each panel item. The GO and STOP panel items are for starting the simulation and pausing the simulation. The RESET panel item is for re-setting all parameters. The QUIT panel item is to kill the simulation and quit the master control panel. The "Show Nodes" panel item consists of a pull-down menu which includes "All Nodes", "Lower Weights", "Upper Weights", and "Output Weights" menu items. Each of menu items is for displaying the present value of nodes during simulation. The "Simulation Status Window" panel item is for showing the current status of simulation. By

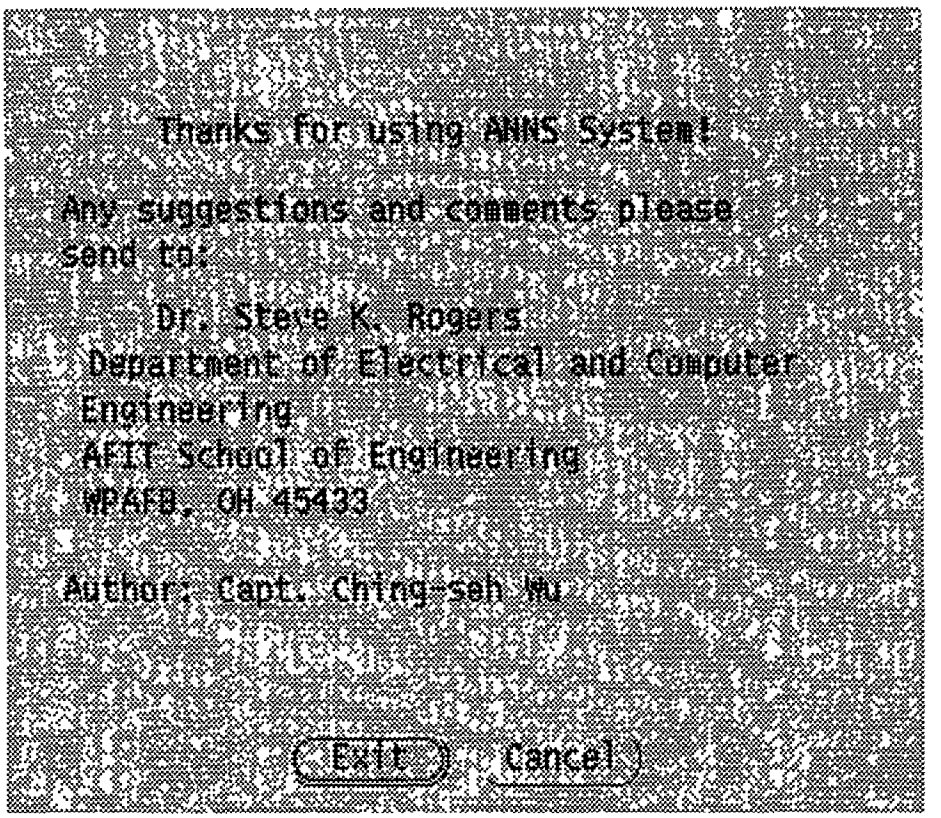


Figure 51. Exit Notification Window

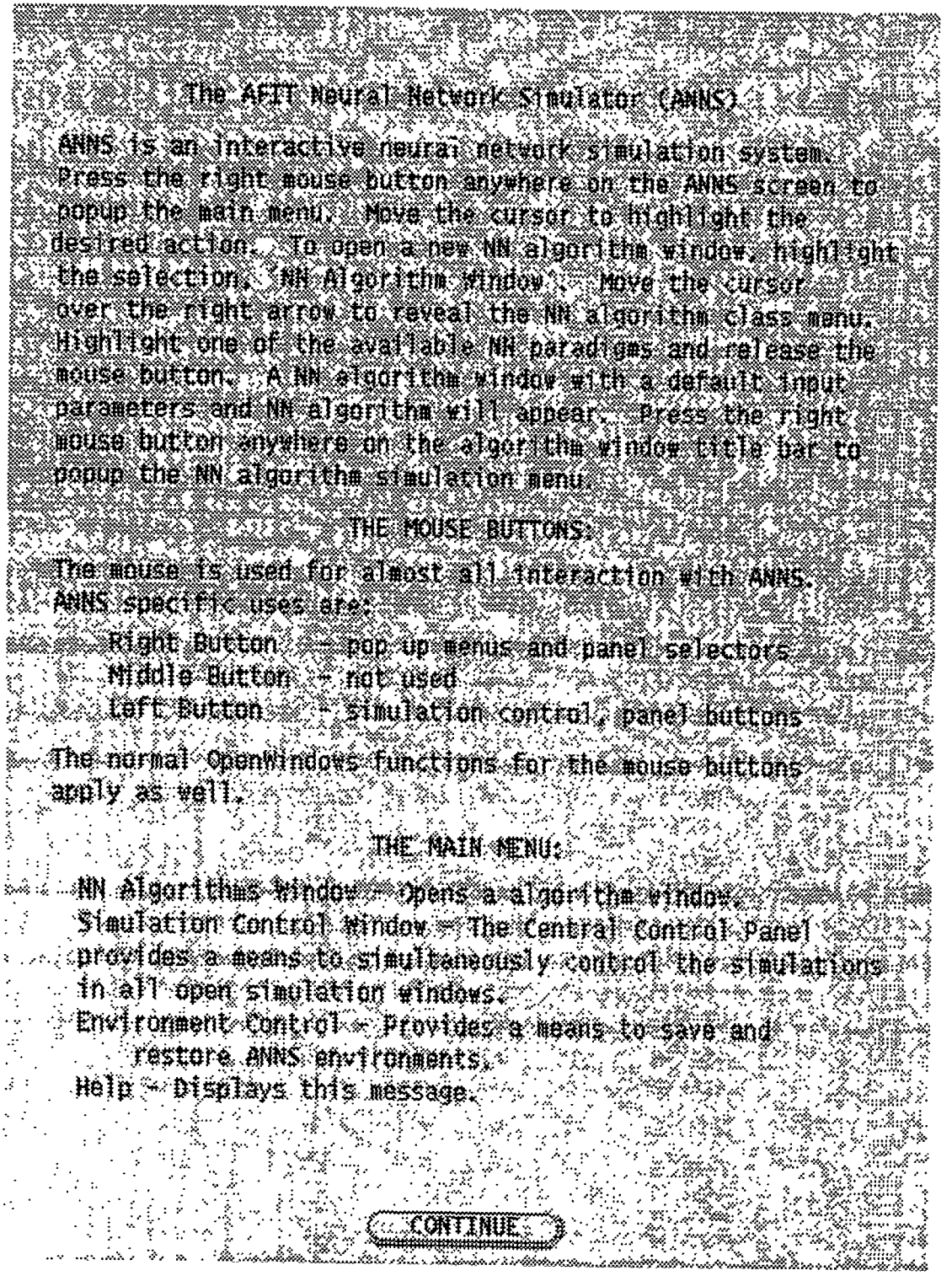


Figure 52. On-line Help Window

clicking the "HIT ME" panel item, the Status Window will appear on screen which displays all the values associated with the currently simulation status. The "Simulation Speed" slide panel item allows user to control the variable simulation speed, or select the "Single Step" panel item for step-by-step simulation.

- **Algorithm Options Section:**

  This section provides the user choice of NN algorithms for simulation. After Changing the NN algorithms, The simulation associated the changed algorithm does not take effect until the RESET panel item is clicked.

- **Error History Window Control Section:**

  This section is for showing the Error History Window or closing the Error History Window. The Error History Window is a graphical window which displays the progress of the network as traing progresses. Also, after each display cycle a tabulation is made to determine the percent correct for both training set and the test set. If the output of the network is within 20 percent of the desired calue for every node, a counter is incremented for the *Right* indicator. If the highest output corresponds to the correct node for that test case the *Good* counter is incremented. These values are displayed for the training and test sets as a percentage correct.

- **Configuration Control Section:**

  This section is for input or change all kinds of parameters associated with the NN algorithm. Different NN apradigms have different configuration parameters.

A.4 Setup

The first step before simulating the NN algorithms is to set up the desired network topology (if the default input parameters are not desired). Select the net type, then type in the number of nodes (computational units) in each hidden layer.

A weight file can be entered at configuration control panel. If there is no previous file enter "r" for random weights to be generated by the system. This allows each input exemplar to rerpresent a cluster of data. Make sure that if you try to restore weights, the weights came from a similar problem. The size of the input and output must be the same as when the weights were stored.

A.4.1 Setup Data Files. The training file contains all the data for testing and training the neural network. The training file name is entered at the prompt of "Data File" panel item on the configuration control panel. Try input files: xor.dat and test.dat. A training file can be created as a standard ascii file using a standard text editor.

The basic format is as follows:

The first line define the number of training vector pairs, the number of test vector pairs, the number of input features, and the number of output classes. After that, each exemplar is listed in order. The first number is arbitrary and ignored by the program. The only purpose is to identify the exemplar number. Next list each element of the exemplar vector x0,x1,x2 . . . x(input-1). The last element is the exemplar class type.

Exemplar class types must be sequential, i.e. 1,2,3,... etc. The first class type must be a one and no numbers can be skipped. Classes can be randomly mixed. To allow flexibility, very little error checking is performed on the input file.

Example:

```
20 20 3 2
1 3.4 5.4 2.1 1
2 5.6 2.3 7.1 0
3 7.2 8.1 6.5 1
4 3.3 4.1 9.1 0
 .
 .
 .
```

```
40 4.1 8.5 3.2  0
```

In the example above there 20 training exemplars and 20 test exemplars. This first class must be labeled zero, the second labeled one etc. The test exemplars are those set aside for testing the generalization capability of the net and are not used in training. The input vector is of length three and there are two distinct output classes.

A.4.2 Backpropagation Paradigm Input Parameter Options. The following parameters adjust the network configuration before training:

- **Saliency:** You have three choices: *Saliency Off, Weight Saliency, or Second Order Saliency.*
- **Output Function:** Choice of *Squashed Output or Linear Output.* The *Squashed output* uses a sigmoid function, and the *Linear output* uses a linear function for node mathematics (for an explanation of node mathematics, see Maj Rogers and ask for a copy of the book: *An Introduction to Biological and Artificial Neural Networks for Pattern Recognition.* [32]
- **Output Format:** Choice of *Class Output, Binary Encode, Vector Output, Identity Output, Identity w/Eigen, Hold One Out, RUCK, and TIME Sequential Data.* In most cases use *Class Output.*
- **Normalization:** Choice of *No Normalization, Statistical Normalization, Energy Normalization, Spread Normalize, Fisher Linear Normalize, Karhunen-Loeve Normalize, Karhunen-Loeve Mean, Principle Components, and Normalize Outputs. Statistical Normalization* forces all inputs to have the same relative ranges and thus is the preferred choice, unless you pre-normalize the data before running Neural Graphics.
- **Layers 1 2 3:** The first entry should be the number of hidden nodes minus one. For example, one hidden layer between input and output requires an entry

of two followed by a space. The next entry should be the number of neurons or nodes as they are called in the first hidden layer. If you have more than one hidden layer, put a space after the first hidden layer number and input the number of neurons for the next hidden layer (moving in the input to the output direction). If you have only one hidden layer, enter the number of nodes in it followed by a space and a zero.

- **Stop At:** To change the total number of iterations (not epochs), enter a number on the keyboard. For example, when configured for 540 training vectors and 60 test vectors, we would have 600 iterations for each epoch. Multiply the number of iterations by 20–50 times to get a first-cut approximation. You will have to experiment to get desired results (a value that causes sufficient training of the data, yet doesn't over train on the data as indicated on the error curve).
- **Weight File:** Click on this panel to input the name of the file that you want to read weight information from. Type the name of the weight file on the keyboard. WARNING: Leave the default file name 'random' under this panel unless you have a weight file that you are importing (in the proper format of course). If the program crashes, check to make sure 'random' is under this panel! Note: A weight file is created every time you train a network; the file created is named 'weights.temp'. ",
- **Data File:** Click on this panel to input the name of the data file you want to train from. Type the name of the file on the keyboard.
- **Statistics:** Click on this panel to input the name of the data file you want to train from. Type the name of the file on the keyboard. Network statistics will be written to this file which are important for plotting the total error versus display updates. ",
- **Average:** This allows user to train through the data several times with new random weights each time (the random generator is reseeded at the start of each training session) and combine the results into an average. Click on this

panel and input a number at the keyboard. Usually we just train once, so input a '1' value.

- **Learning Rate:** Leave set to '1'; the code is 'hard-wired' to 1 over(no. nodes fan-in).
- **Display:** This number represents how often the screen is updated with new information during training and testing. To change this number, type the new number at the keyboard. (start with about '1000').
- **Noise:** Click on noise if you want to add noise to the data. A number greater than zero will add noise (i.e., noise = 1 gives a random flat distribution). Zero indicates no noise. As a suggestion, do not use noise on your first training attempt.

A.5 Run simulation

Before a simulation is running, default values are supplied. To accept a default, just hit the GO panel to start simulating the selected NN algoriuthm. The simplest way to learn about the ANNS is just to accept all the defaults input parameters which usually show up on the configuration control panel. Just hit GO panel, that should get the default model running with reasonable default configurations. Try testing each of the NN algorithms on the NN algorithm menu to see what happens. Try re-running the program with values other than the defaults. Use different sized nets, different numbers of hidden layers, different weight files, different data files and hit RESET panel and then GO panel to restart simulation.

Once, you feel comfortable making up your own net configurations, try some real work. Make the computer learn something. You might want to start a multilayer perceptron model, usually call backprop. Backprop comes in a variety of flavors, the models used here are two of the most commonly used, backprop and conjugate gradient.

Appendix B. ANNS Programmer's Guide

B.1 Introduction

The AFIT Neural Network Simulator (ANNS) is a development and tutorial system for the study and research of artificial neural networks. It is also an integrated and graphically interactive neural network paradigm simulation system. ANNS allows the end-users to select the optional neural network paradigms and algorithms for simulation, and allows the client-programmers to add new neural network paradigms and algorithms to ANNS. **This manual is intended for client-programmers;** it describes the overall ANNS implementation and provides a general guide for adding new Neural Network (NN) paradigms into ANNS. Before doing that, Client-programmers should also refer to the ANNS User's Manual for better understanding of the operations of ANNS. End-users should refer to the ANNS User's Manual.

The ANNS system is intended to test feature extraction methods, compare training paradigms, and help the user understand the limitations and utility of this novel approach to computing. ANNS runs on Sun SPARCstastions using the SunOS (Sun Microsystem's version of the AT&T UNIX operating system) and *OpenWindow* environment at Air Force Institute of Technology (AFIT). There are several control mechanisms provided, including GO simulation, STOP simulation, RESET all parameters, variable simulation SPEED, single simulation STEP, and QUIT the simulation. Other features include:

- a NN paradigm class selection from a list of available classes stored in the paradigm library directory, such as *back-propagation, hybrid training, radial basis function, hopfield associative memory, etc..*
- Dynamical simulations with color monitors.
- Multiple NN simulation windows, each of which with one master control panel.

- Master control panel for modifying the input parameters and control parameters, and for monitoring the simulation status associated with each NN paradigm.
- Simulation environment save and restore capability.
- Simultaneous comparison and control of multiple Neural Network (NN) paradigm and algorithm simulations.
- Provide on-line help for every interface function and window.
- Provide error window to illustrate the error surface.
- provide simulation status window to show the static simulation status at each training step.

B.2 Backgrounds Needed for Programmers

This section provides some recommendations for client-programmers to strengthen the needed backgrounds for ANNS.

First of all, a client-programmer should have experience with ANNS as end-users and be familiar with the terms and concepts associated with artificial neural networks, then the following references are recommended:

- *Programming with C [8]:* Everything was written in C programming language for ANNS.
- *UNIX Operating System:* ANNS is running on UNIX operating system environment.
- *Introduction to the X Window System [15]:* What is X Window System ?
- *XView Programming Manual [9]:* The Graphical User Interface (GUI) of ANNS was implemented using XView widget sets.
- *SunView Programmer's Guide [47]:* Some of the graphical routines of XView were derived from SunView.

- *Network Programming [43]:* The concepts of using UNIX sockets for integrating the ANNS system.
- *SunOS Reference Manual [46]:* ANNS is running on Sun Microsystem which are controlled by SunOS.

B.3 Overview of ANNS

This section provides an introduction to several topics of interest to ANNS programmers and NN paradigm simulation developers.

B.3.1 Objects Associated with X Window System. This section provides an object model (associated with ANNS system design) of the X Window System, as an example to illustrate the concepts of object-oriented design and X Window System. Figure 53 describes many object modeling constructs and shows how they fit together into a large model.

Class *Window* defines common parameters of all kinds of windows, including a rectangular boundary defined by the attributes *x1, y1, x2, y2,* and operations to display and undisplay a window and to raise it to the top (foreground) or lower it to the bottom (background) of the entire set of windows. *Panel, Canvas,* and *Text window* are varieties of windows. A canvas is a region for drawing graphics. It inherits the window boundary from *Window* and adds the dimensions of the underlying canvas region defined by attributes *cx1, cy1, cx2, cy2.* A canvas contains a set of elements, shown by the association to class *Shape.* All shapes have color and line width. Shapes can be lines, ellipses, or polygons, each with their own parameters. A polygon consists of an ordered list of vertices, shown as an aggregation of many points. Ellipses and polygons are both closed shapes, which have a fill color and a fill pattern. Lines are one-dimensional and cannot be filled. Canvas windows have operations to add elements and to delete elements.

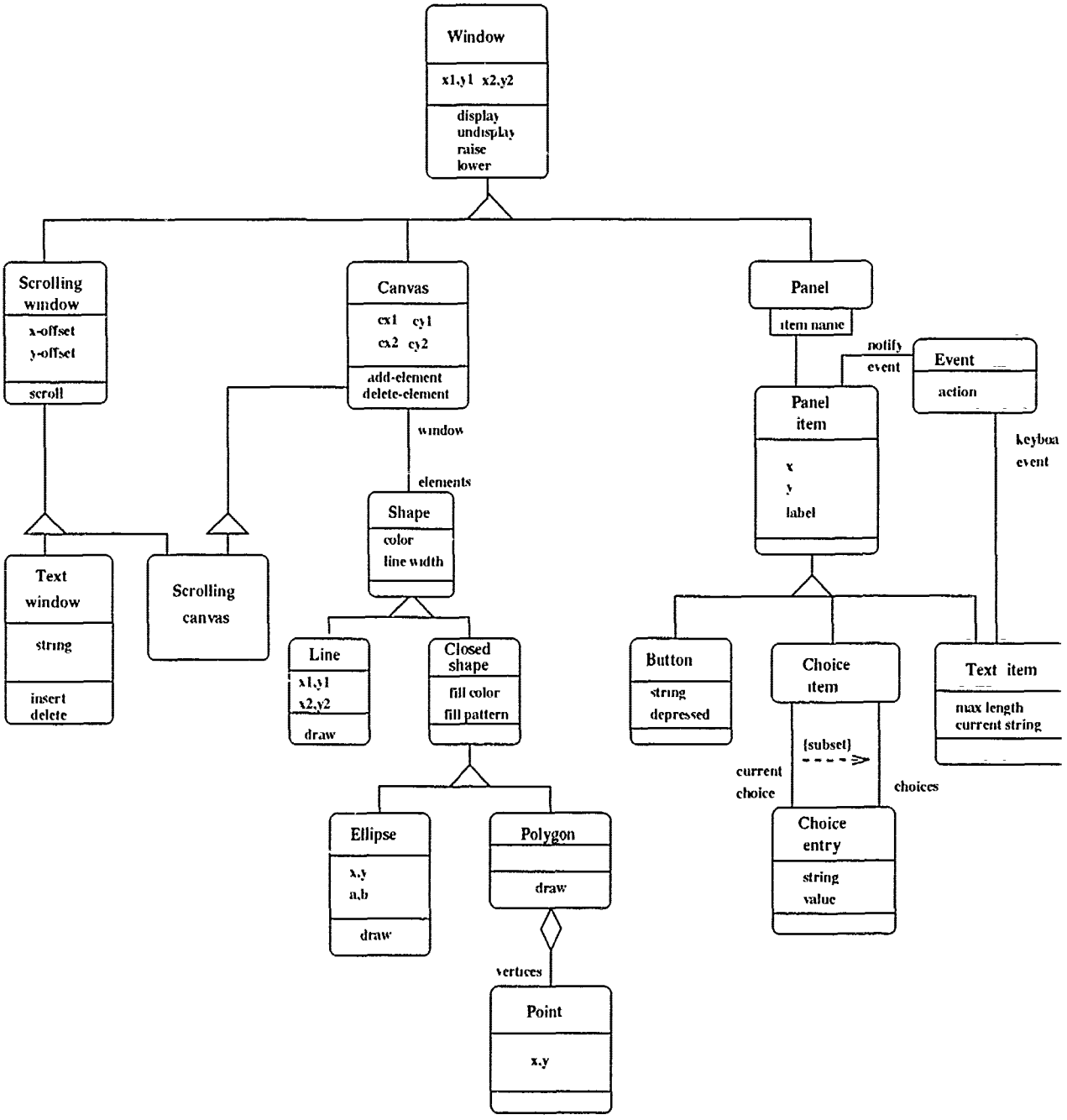


Figure 53. Object Model of X Window System

Text window is a kind of a *Scrolling window*, which has a 2-dimensional scrolling offset within its window, as specified by *x-offset* and *y-offset*, as well as an operation *scroll* to change the scroll value. A text window contains a string, and has operations to insert and delete characters. *Scrolling canvas* is a special kind of canvas that supports scrolling; it is both a *Canvas* and a *Scrolling window*. This is an example of *multiple inheritance*.

A *Panel* contains a set of *Panel item* objects, each identified by a unique *item name* within a given panel, as shown by the qualified association. Each panel item belongs to a single panel. A panel item is a predefined icon with which a user can interact on the screen. Panel items come in three kinds: buttons, choice items, and text items. A button has a string which appears on the screen; a button can be pushed by the user and has an attribute *depressed*. A choice item allows the user to select one of a set of predefined *choices*, each of which is *Choice entry* containing a string to be displayed and a value to be returned if the entry is selected.

When a panel item is selected by the user, it generates an *Event*, which is a signal that something has happened together with an action to be performed. All kinds of panel items have *notify event* associations. Each panel item has a single event, but one event can be shared among many panel items. Text items have a second kind of event, which is generated when a keyboard character is typed while the text item is selected. Association *keyboard event* shows these events. Text items also inherit the *notify event* from superclass *Panel item;* the *notify event* is generated when the entire text item is selected with a mouse.

B.3.2 ANNS Architecture. ANNS uses three level of execution linked by UNIX sockets to implement the NN simulation system. Figure 54 shows the idealistic abstract model of ANNS and figure 55 shows the architecture of ANNS. Sockets are the InterProcess Communication (IPC) mechanism provided by the UNIX operating system. Each level is a self-contained, executable system. Without the sockets, each

individual class system in the ANNS system could be run stand-alone. The three levels of execution are:

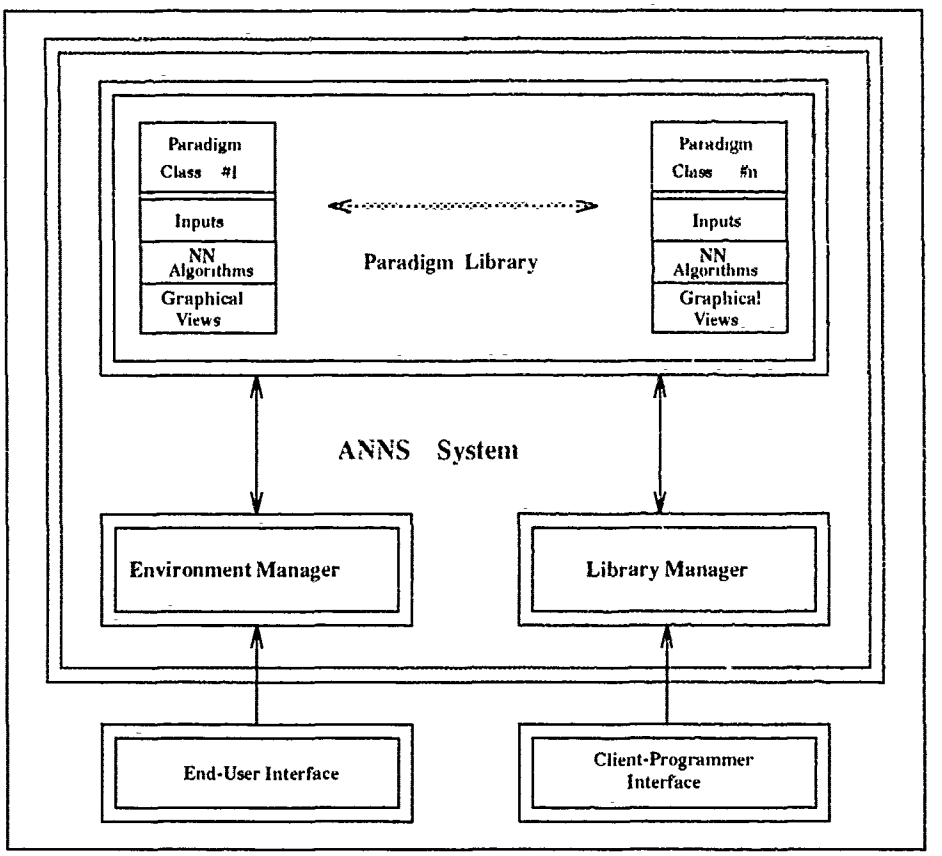


Figure 54. The Idealistic Abstract Model of ANNS

- *The ANNS main process.*

  The ANNS main process is the highest level interface to ANNS. It acts as the environment manager and creates main process window, the main menu window, the central control window, the environment control window, exit window, help window and a means for starting NN paradigm simulations.

- *The ANNS common library.*

  The middle level of execution is the NN paradigm-specific window-based process called ANNS common library. Each NN paradigm component interfaces

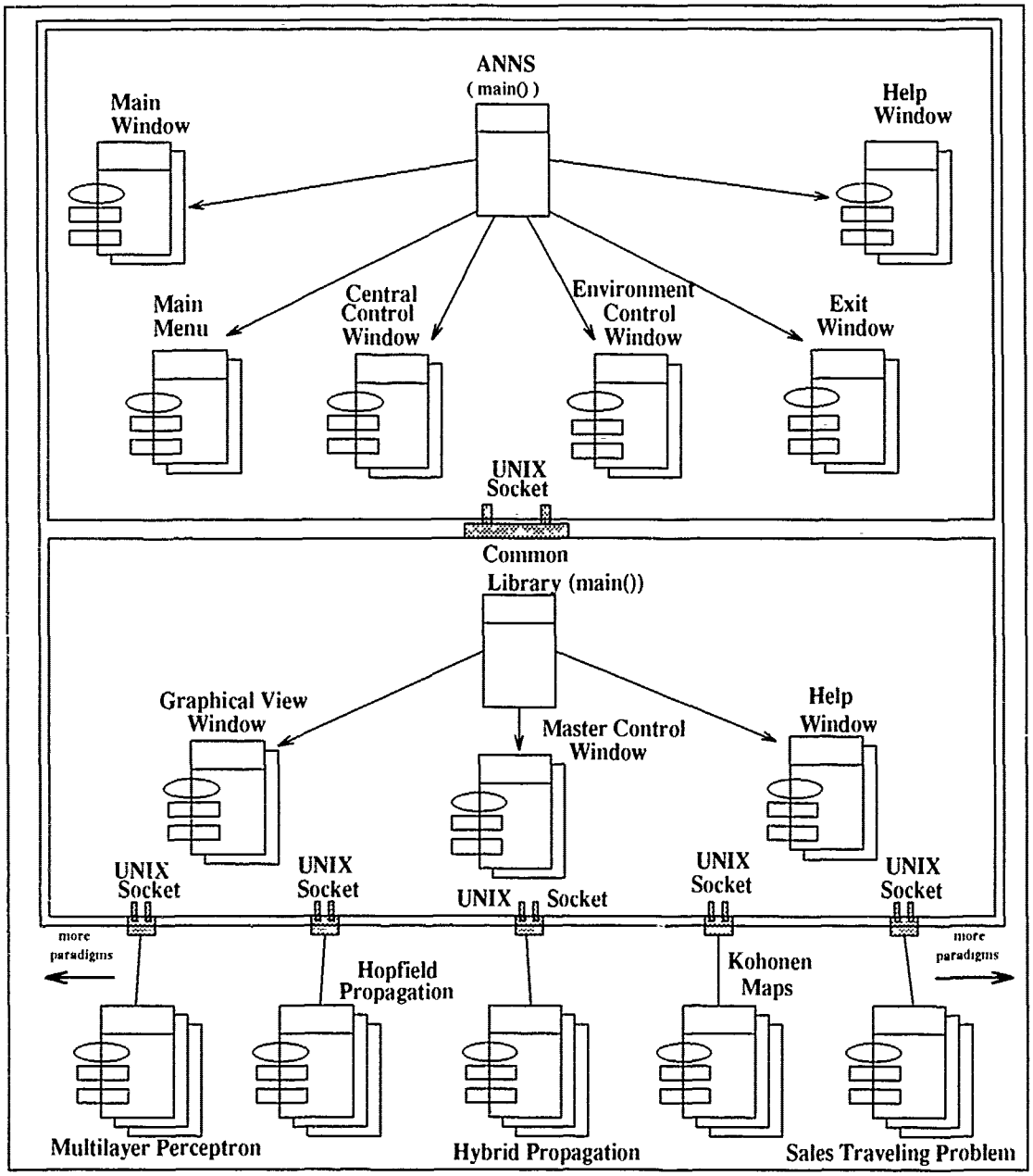


Figure 55. The Architecture of ANNS at the Top Level

with the common library for simulatior control. The common library provides: the graphical view windows for viewing the dynamically graphical display, the master control window for monitoring and modifying the parameters that affect the simulation, and the online help window for understanding the use of all the parameters in the master control window.

- *The ANNS paradigms.*

  This is the lowest level of execution which is transparent to the users. Each NN paradigm has three window associated with the ANNS common library. The graphical view window is directly under the paradigm's control. There is only one NN paradigm (multilayer feedforward networks using backpropagation) implemented so far, including eleven algorithms: Back Propagation, Back Prop W/Momentum, Second Order Learning, Cotrell Identity Net, Tarr/Cotrell Identity, Auto-Add a Layer, Gram_Schmidt Network, Gram-Schmidt ID Net, BrainMaker, Radial Basis, and Conic Basis.

B.3.3 Object Creating and Mapping Using XView.

B.3.3.1 Objects in XView. There are eight basic object types in XView. Three of these, *Generic Objects*, *Windows*, and *Openwins* are core classes and are not directly instantiable by the user. The remaining five are discussed below. The basic window entity is the *frame*. All other windows are classified as *subwindows* and must be attached to frames.

- **Frames**

  A frame is the basic window object to which the programmer has access. There are two flavors, a base frame, and a pop-up frame. A base frame is a frame with no parent. All other frames are subframes, so a pop-up is any frame which is not the base frame. Each application has one base frame. There are no (preset) limits on the number of subframes. A frame is characterized by a border, which is managed by the window manager, and an interior which is

configured and managed by the programmer. The window manager controls resizing, iconification, de-iconification, refreshing, quitting, etc. All XView windows are framed.

- **Canvases**

  A *canvas subwindow* is the XView graphics window. It's size is independent of the owning frame. The entire drawing surface is called the *paint window* and the visible portion is the *view window*. Multiple, scrollable views of a canvas are allowed within a frame.

- **Text Windows**

  A *text subwindow* provides basic text editing capabilities using the OPEN-LOOK text editing model. This window is a specialization of the canvas subwindow with text editing capabilities added.

- **Menus**

  Menus are not actually XView windows, but they are bound to windows at display time. XView supports a full range of menu types and options such as pull-down, pop-up and pull-right. Menus can be *pinned* to allow them to stay on the screen after the selection is made.

- **Scrollbars**

  Scrollbars are interesting objects. They can exist independently, or be attached to subwindows. Scrollbars are windows (because they are visible) but they are usually thought of as properties of subwindows. Scrollbars do not manage the objects to which they are attached, it is the programmer's responsibility to make the screen updates associated with scrollbar actions.

An important feature of XView that is not a window is the *Panel*. Panels implement the OPENLOOK *control area*. Panels manage *panel items*, e.g. buttons, sliders, text fields, and other forms of inputting data. The motivation for panels is to provide a mechanism for propagating events, especially for objects which do not have a window associated them. Panels are very important in XView. For example, an application

frame with no subwindows attached cannot catch interior mouse events. Attaching a panel to the frame allows these interior events to be propagated.

B.3.3.2 Object Creating and Mapping. XView provides a very clean interface to it's object set. There is a common set of functions that allows users to manipulate any objects by referencing the object handle. The functions are:

- **xv_init():**

  Establishes the connection to the server, initializes the Notifier and the Defaults/Resource-Manager database, loads the Server Resource Manager database, and parses any generic toolkit command line options. Called once at the beginning of the program.

- **xv_create():**

  Creates an object. Every XView object is created with this function.

- **xv_destroy():**

  Destroys an object.

- **xv_find():**

  Searches for and returns an object with the specified parameters. If none is found, the object is created.

- **xv_get():**

  Get the value of the specified attribute for the specified object.

- **xv_set():**

  Set the value of the specified attribute for the specified object.

Using these six routines, programmers can create and manipulate the entire XView object set for ANNS.

B.3.4 ANNS Directory Structure. So far, the ANNS main directory structure consists of several subdirectories which are (see figure 56):

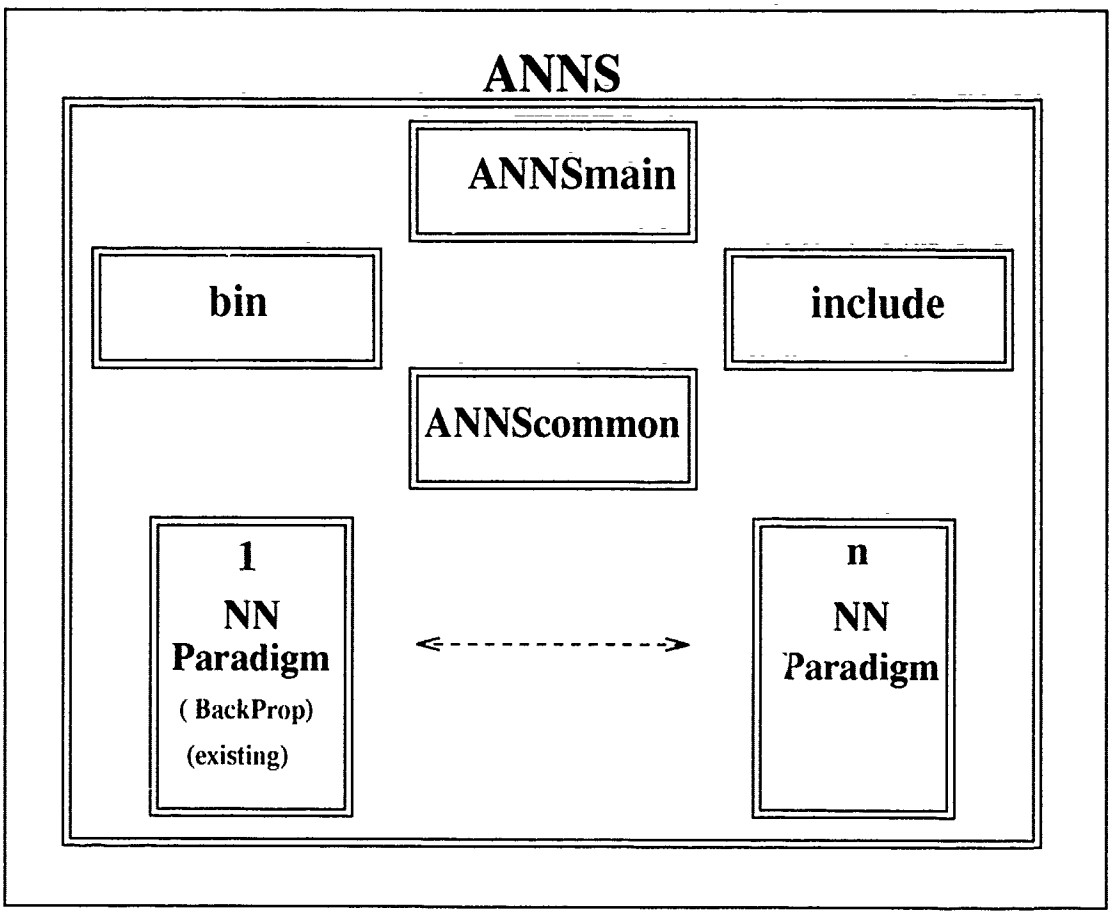


Figure 56. ANNS Directory Structure

B.3.4.1 ANNSmain Subdirectory. This sundirectory includes all the source codes for the ANNS main process. It provides an environment in which all NN paradigm simulations can be controlled. The files in this directory are:

- *ANNSmain.c*

  creates the ANNS main window and ANNS icon image, sets up the control environment, and processes the mouse events.

- *ANNSUtility.c*

  contains subprogram modules to get the screen resolutions, dispatch the window events, filter the input characters, get the directory of input files, process user warning cases, and create a color table.

- *ANNSmainmenu.c*

  creates ANNS main menu.

- *ANNScontrol.c*

  contains the functions for setting up and controlling a centralized simulation control panel. This control panel is in ADDITION to the usual simulation control provided by the specific NN algorithm class. The panel is OFF by default. Activation has no effect on simulations other than providing another means of control.

- *ANNSenvironment.c*

  contains the subprogram modules to create ANNS Environment Panel and to save and restore environments.

- *ANNSwindows.c*

  contains the functions for defining, opening, and closing NN algorithm windows.

Each of these modules above has an individual header file with it for controlling global parameters.

B.3.4.2 ANNScommon Subdirectory. This subdirectory presents an interface for adding NN paradigms to ANNS by providing all the common modules for each of NN paradigms. After compiling all the files in this directory, an ANNS common library archive file called **libANNS.a** is created in the **bin** subdirectory for next level of compiling and execution. The NN paradigm class simply provides the simulation Interesting Events (IEs) which are operation components of the simulation to connect to this common library. Figure 57 shows the communications of the IEs between a NN paradigm class and this ANN common library.

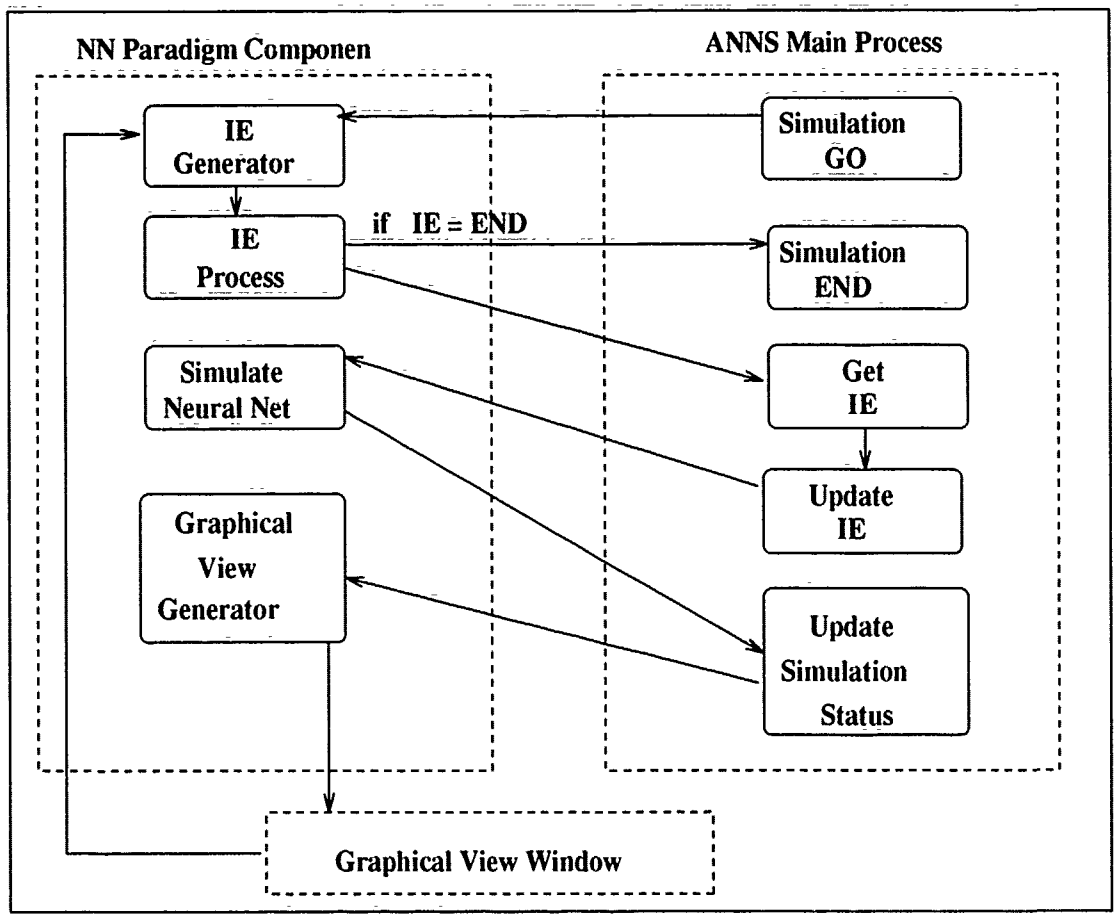


Figure 57. The IEs Communication Model

The files in this directory are:

- *ANNScommon.c*

  is a basic shell for all NN algorithm simulation packages. Each NN paradigm simulation links to this file. This file provides the main routine, establishes the communications link to ANNS, creates the master control panel, simulation status window, error history window, and simulation canvas. Client-Programmers simply provide NN algorithm, and operation components of the simulation (IEs) which link to these functions.

- *ANNSparameters.c*

  contains all subprogram modules to read simulation parameters, save simulation parameters, restore simulation parameters, and send all the simulation parameters to ANNS. All parameters depend on the NN paradigm class-specific interesting event structure (IE_STRUCT). IE_STRUCT is referred to as an ADT, but its internal structure is not important. The NN paradigm class-specific functions must provide a macro or a function to copy IE_STRUCTs.

- *ANNSmaster.c*

  contains subprogram modules to create master frame, set up control section of the master control panel, set up simulation status window, and get all the control parameters.

- *ANNSgraphicalView.c*

  contains subprogram modules to set up the simulation canvas, handle IEs, simulate NN algorithm, control simulation speed, control multiple graphical view window.

- *ANNSUtility.c*

  contains subprogram modules to get the screen resolutions, dispatch the window events, filter the input characters, get the directory of input files, process user warning cases, and create a color table.

Each of these modules above also has an individual header file with it for controlling global parameters.

B.3.4.3 **bin** *Subdirectory.* This subdirectory contains all the executable files, archive files (common library), and one ANNS NN paradigm class file called *.annsParadigms*. The *.annsParadigms* file lists all the available NN paradigm components and the names of executable files for each NN paradigm component. Client-programmers must add the name of the new NN paradigm component and the name of the executable file (associated with this new NN paradigm component) to *.annsParadigms* file. ANNS need not be recompiled if this file is modified. This file will be called when the ANNSmainmenu.c is executed. The *.annsParadigms* file is shown as follow:

```
Multilayer Perceptron Network,      /home/hawkeye3/cwu/ANNS/bin/Test
Hopfield Net ,                     /home/hawkeye3/cwu/ANNS/bin/HopNet
Hybrid Propagation,               /home/hawkeye3/cwu/ANNS/bin/HybNet
Kohonen Maps,                    /home/hawkeye3/cwu/ANNS/bin/Kohonen
Salese Travel Problem,         /home/hawkeye3/cwu/ANNS/bin/TSP
~

    /****** Format for Above ********/
(1) List the name of a new NN paradigm, followed by a comma, and the name of
    the executable file. Only one NN paradigm per line
(2) After tilde "~" , nothing is read
/*************************************************************************
 *               AFIT Neural Network Simulator (ANNS)
 * DATE :           13 Feb 1993
 * VERSION:         1.0
 * NAME :           .annParadigms
 * MODULE NUMBER:
 * DESCRIPTION :    This file consists of a list of the NN algorithm
 *                  components available to ANNS
 *                  There are two lists:
 *                  (1).paradigmName = The name of each NN paradigm; used
 *                                     in menus and as the window titles
 *                  (2).executableFile = The name of an executable file
 *                                     for the specific NN paradigm; used by
 *                                     ANNS to execute a simulation.
 *    The Client-Programmer must update this file when a new NN paradigm is
 *    added to the ANNS system.  ANNS need not be recompiled if this file is
 *    modified.
 * ALGORITHM:
 * PASSED VARIABLES:  None
 * RETURNS:           Nothing
 *************************************************************************/
```

B.3.4.4 **include** *Subdirectory.* This subdirectory contains header files, and icon image files for ANNS.

- *annsDefines.h* contains all included system header files and predefined constants.

- *commonDefines.h* contains defines and types shared by all the "paradigmCommon" and "paradigmSpecific" functions.

B.4 General Procedure for Adding a New NN Paradigm

A existing NN paradigm class is probably the best guide for adding a new NN paradigm to ANNS. Client-programmers may take the existing NN paradigm directory, like *BackProp*, as an example. Assuming that a client-programmer has a NN paradigm coded in C language (graphical routines was coded using XView), the following sections are suggested procedure for adding the NN paradigm to ANNS:

B.4.1 Create a working directory. There is a template directory, called *newParadigm*, in the ANNS main directory. Use this directory as a guide for development by copying this directory and changing the name of this directory to the new paradigm's name under the ANNS main directory. There are totally seven files in this template subdirectory: *Makefile*, *NewParadigm.h*, *BG_main.c*, *Algorithms.c*, *GraphicalView.c*, *and Control.c*, *and Initial.c*Besides these files, add all the source files of the new NN Paradigm to this directory.

1. *Makefile* is an utility for lowest level of compiling. The example of *Makefile* is as following:

```
#************************************************************************
#  FILENAME : Makefile
#  DESCRIPTION : Makefile for the New NN paradigm
#************************************************************************

ParadigmNAME = NewParadigm

#************************************************************************
#Set the base path to the directory in which the executable will be stored
#************************************************************************
BASE          = $(HOME)/ANNS
BIN           = $(BASE)/bin
INCLUDE       = $(BASE)/include
ParadigmPRG   = $(BIN)/$(ParadigmNAME)
ParadigmBG    = $(BIN)/$(ParadigmNAME)BG

LIBS = -Bstatic -lnet -lANNS -lxview -lolgx -lX11 -lsuntool -lsunwindow  -lpixrect -lm

CC = cc
```

```
CFLAGS = -I$(INCLUDE) -I$(OPENWINHOME)/include -L$(OPENWINHOME)/lib

LDFLAGS = -L$(BIN)

DEPFLAGS = -MM $(CFLAGS)

#*************************************************************************
# NewParadigm files
#*************************************************************************

# add New NN paradigm object files here --like the Backprop paradigm
# NewParadigmOBJS = GraphicalView.o Algorithms.o Control.o Initial.o
NewParadigmOBJS = GraphicalView.o Algorithms.o Control.o Initial.o backprop.o \
                  backprop2.o backprop3.o radial.o backprop8.o conjugate.o \
                  preprocess.o batch.o cluster.o backprop4.o backprop5 o \
                  backprop6.o backprop7.o poly.o random.o normal.o test.o \
                  jacobi.o nrutil.o eigsrt.o invert.o lubksb.o ludcmp.o\
                  gaussj o initialize.o makeinput.o paradigm.o saver o \
                  general.o display.o graphic.o

# add New NN paradigm files here --like the Backprop paradigm
#NewParadigmSRCS = GraphicalView.c Algorithms.c Control c Initial.c
NewParadigmSRCS = GraphicalView.c Algorithms.c Control.c Initial.c backprop c \
                  backprop2.c backprop3.c radial.c backprop8.c conjugate.c \
                  preprocess.c batch.c cluster.c backprop4 c backprop5 c \
                  backprop6.c backprop7.c poly.c random.c normal.c test.c \
                  jacobi.c nrutil.c eigsrt c invert.c lubksb.c ludcmp c\
                  gaussj.c initialize c makeinput.c paradigm.c saver.c \
                  general.c display.c graphic.c


BGOBJS = BG_main.o

BGSRCS = BG_main.c

SRCS = $(NewParadigmSRCS) $(BGSRCS)

OBJS = $(NewParadigmOBJS) $(BGOBJS)

#*************************************************************
# Compiling rules
#*************************************************************


$(ParadigmNAME): $(ParadigmPRG) $(ParadigmBG)

$(ParadigmPRG):  $(NewParadigmOBJS) $(BIN)/libANNS.a $(BIN)/libnet.a
$(CC) $(CFLAGS) $(LDFLAGS) -o $(ParadigmPRG) $(NewParadigmOBJS) $(LIBS)

$(ParadigmBG):  $(BGOBJS)
$(CC) $(CFLAGS) $(LDFLAGS) -o $(ParadigmBG) $(BGOBJS) $(LIBS)

Control o: Control.c Makefile
$(CC) $(CFLAGS) -c Control.c -DBGFILE='"$(ParadigmBG)"'
```

2. *NewParadigm.h*

   is a header file for defined global parameters and defined constants (add what the New Paradigm has to this file).

3. *BG_main.c*

is the main background procedure that drives the graphical display by selecting one of the algorithms and delivering interesting event (IE) announcements to the evoking routine on request. The example file is as following:

```
/*****************************************************************************
 *                  AFIT Neural Network Simulator (ANNS)
 * FILENAME     : BG_main.c
 * FUNCTIONS:
 *     main() - establishes IPCs.  Accepts parameter
 *         structure from the main algorithm routine.
 *     INTERESTING_EVENT() - IPC function; sends the IE operations to the
 *         main algorithm routine, after a request has been received.
 *
 *****************************************************************************/
#include <sys/types.h>
#include <sys/socket.h>
#include "NewParadigm.h"
/*****************************************************************************
 * FUNCTION NAME    : main()
 * DESCRIPTION:
 *     Evoked by ANNS main process.
 *****************************************************************************/
main(argc, argv)
int argc,
char *argv[],
{
  int    socket;
  void    INTERESTING_EVENT();
  INIT_PAK parameters;
  int count=0,stopit=10000;
  int avg,avgs=50,

  socket = atoi(argv[1]);

  if(read(socket, &parameters, sizeof(INIT_PAK)) < 0)
  {
    perror("BG(1) reading control parameters'); exit(23);
  }
/**** All algorithms for BackProp paradigm component*****/
   switch(parameters.algorithm[P_ALGORITHM])
    {
        case BACKPROP:
            init_paradigm = init_train_BACKPROP;
            break;
        case BACKPROP2:
            init_paradigm = init_train_BACKPROP2;
            break;
        case BACKPROP3:
            init_paradigm = init_train_BACKPROP3;
            break;
        case BACKPROP6:
            init_paradigm = init_train_BACKPROP6;
            break,
        case BACKPROP8:
            init_paradigm = init_train_BACKPROP8;
            break;
        case BACKPROP9:
            init_paradigm = init_train_BACKPROP9;
            break;
```

```
        case BACKPROP10:
            init_paradigm = init_train_BACKPROP10;
            break;
        case BACKPROP_GS:
            init_paradigm = init_train_BACKPROP_GS;
            break;
        case BACKPROP_GSID:
            init_paradigm = init_train_BACKPROP_GSID;
            break;
        case BACKPROPC:
            init_paradigm = init_train_BACKPROPC;
            break;
        case RADIAL:
            init_paradigm = init_train_RADIAL;
            break;
        case preprocess:
            init_paradigm = init_train_preprocess;
            break;
        case BRAIN:
            init_paradigm = init_train_BRAIN;
            break;
        default:
            perror(" Unknown algorithm for BackProp Paradigm"); exit(25);
    }
/*** display the title "Multilayer Perceptron"***/
INTERESTING_EVENT(socket, INIT);

for(avg=1;avg<=avgs;avg++){
INTERESTING_EVENT(socket, Initialize_Net);

    for(count=0;count <stopit;count ++){
      INTERESTING_EVENT(socket, Make_Input);
      INTERESTING_EVENT(socket, Propagation);
      INTERESTING_EVENT(socket, Train_Net);
      INTERESTING_EVENT(socket, Display_Net);
      }
      INTERESTING_EVENT(socket, FINAL);
    }
    INTERESTING_EVENT(socket, DONE);

  INTERESTING_EVENT(socket, BG_WAIT);
}
/*************************************************************************
 * FUNCTION NAME   : void INTERESTING_EVENT(int, int, int, int)
 * DESCRIPTION:
 *     Receives an Interesting Event argument, waits on a
 *     IErequest from the ANNS main process, sends the IE operations.
 * INPUT PARAMETERS:
 *     socket - plug in main process
 *     type - type of event
 * OUTPUT PARAMETERS:
 *     write IE_DATA to main algorithm process
 *************************************************************************/
void INTERESTING_EVENT(socket, type)
int socket, type;
{
  IE_DATA IEpacket;
      int IErequest;

  if(read(socket, &IErequest, sizeof(int)) < 0)
  {
    perror("Waiting for IErequest"); exit(20);
  }
  if(IErequest == BG_QUIT)
    exit(0);
```

```
  IEpacket.type   = type;
  if(write(socket, &IEpacket, sizeof(IE_DATA)) < 0)
  {
    perror("Writing IE operations"); exit(22);
  }
 }
```

4. *Algorithms.c*

contains functions which manage the NN algorithm parameters for the NN new paradigm class of algorithms. The example file is as following:

```
/*************************************************************************
 * FILE : Alg.c
 * DESCRIPTION : This file contains functions which manage the NN
 *     algorithm parameters for the BackProp class of algorithms.
 * FUNCTIONS :
 *      int  addAlgorithmSection(Panel, int)
 *     void  getAlgorithmParameters(PARAMS)
 *     void  setAlgorithmParameters(PARAMS)
 *     char *setAlgorithmName()
 *************************************************************************/
#include "NewParadigm.h"
static Panel_item algorithmItem;
/*************************************************************************
 * FUNCTION NAME : int addAlgorithmSection(Panel, int)
 * DESCRIPTION : Adds to the Master Control Panel the items that
 *     control algorithm parameters.
 * INPUT PARAMETERS :
 *     panel - the panel to which the items are added
 *     row - the panel row on which the items begin
 * OUTPUT PARAMETERS :
 *     row - the panel row on which the next panel item should begin
 *************************************************************************/
int addAlgorithmSection(panel, row)
Panel panel;
  int row;
{
    (void) xv_create(panel, PANEL_MESSAGE,
PANEL_NEXT_ROW, -1,
PANEL_LABEL_BOLD, TRUE,
        PANEL_LABEL_STRING,
            "---------------------- ALGORITHM OPTIONS -----------------",
        NULL);

    algorithmItem = xv_create(panel, PANEL_CHOICE,
PANEL_NEXT_ROW, -1,
PANEL_DISPLAY_LEVEL, PANEL_CURRENT,
        PANEL_LABEL_STRING,     "Algorithm Type :",
        PANEL_CHOICE_STRINGS,   "Back Propagation",
                                "Back Prop W/ Momentum",
                                "Second Order Learning",
                                "Cotrell Identity Net",
                                "Tarr/Cotrell Identity",
                                "Auto-Add a Layer",
                                "Gram_Schmidt Network",
                                "Gram-Schmidt ID Net",
                                "BrainMaker",
                                "Radial Basis",
                                "Conic Basis",
                                 NULL,
        PANEL_VALUE,             0,
```

```
        NULL);
        return(row+4);
}
/*************************************************************************
 * FUNCTION NAME : void getAlgorithmParameters(PARAMS)
 * DESCRIPTION : Saves the current algorithm parameters.
 * INPUT PARAMETERS . parameter - PARAMS array
 * OUTPUT PARAMETERS : None
 * GLOBALS USED : algorithmItem
 * FUNCTIONS CALLED: xv_get()
 *************************************************************************/
void getAlgorithmParameters(parameter)
PARAMS parameter;
{    int algorithm;

        algorithm = (int)xv_get(algorithmItem, PANEL_VALUE);
    parameter[P_ALGORITHM] = (int)xv_get(algorithmItem,  PANEL_VALUE);
}
/*************************************************************************
 * FUNCTION NAME : void setAlgorithmParameters(PARAMS)
 * DESCRIPTION : Sets the algorithm paramters after a restore operation.
 * GLOBALS USED : algorithmItem
 * GLOBALS AFFECTED : sets value of algorithmItem
 * FUNCTIONS CALLED: xv_set()
 *************************************************************************/
void setAlgorithmParameters(parameter)
PARAMS parameter;
{
    xv_set(algorithmItem, PANEL_VALUE, parameter[P_ALGORITHM], NULL);
}

/*************************************************************************
 * FUNCTION NAME : char* getAlgorithmName()
 * DESCRIPTION : Provided so that the common setALgWindowTitle() function
 *     can get the algorithm name without any global naming conventions.
 * INPUT PARAMETERS : None
 * OUTPUT PARAMETERS : pointer to string (algorithm name)
 * GLOBALS USED : algorithmItem
 * GLOBALS AFFECTED : None
 * CALLED BY : setAlgWindowTitle()
 * FUNCTIONS CALLED: xv_get(), panel_get()
 *************************************************************************/
char* getAlgorithmName()
{
    int algorithm;

    algorithm = (int)xv_get(algorithmItem, PANEL_VALUE);
    return((char *)xv_get(algorithmItem, PANEL_CHOICE_STRING, algorithm));
}
```

5. *GraphicalView.c*

processes IEs, paints all graphical views on simulation canvas, and updates the simulation status. This is the most important file for Client-programmers. The functions that client-programmers need to modify are *addConfigSection()*, *updateNet()* and *updateStatus()*. The example of the *updateNet()* function that

need to be modified is as following (Compare this functions with *BG_main.c* for understanding of the IE processes.):

```
/*************************************************************************
 * FUNCTION NAME : void updateNet(int, int, Pixwin)
 * DESCRIPTION : Updates the graphical view
 *************************************************************************/
extern nnet *net;
extern int which_exemplar;
extern setup BPparameters;
void updateNet(type,thePixwin)
int type;
Pixwin *thePixwin;
{
  switch(type)
    {
    case INIT:
        /**** CLEAR THE CANVAS ****/
            pw_rop(thePixwin, 0, 0, NetWidth,  NetHeight,
                PIX_SRC | PIX_COLOR(WHITE), NULL, 0, 0);
           pw_text(thePixwin,5, 10,
                PIX_SRC | PIX_COLOR(BLUE), NULL, "Multilayer Neural Net");
         break;
/***These cases call NN operation functions*****/
    case Initialize_Net:
         INITIALIZE(TRUE);
         break;
    case Make_Input:
         MAKE_INPUT(net->inp, net->doft,which_exemplar);
         break;
    case Propagation:
         FEED_FORWARD();
         break;
    case Train_Net:
         TRAIN_NET();
         break;
    case Display_Net:
        /**** CLEAR THE CANVAS ****/
            pw_rop(thePixwin, 0, 0, NetWidth,  NetHeight,
                PIX_SRC | PIX_COLOR(WHITE), NULL, 0, 0);
           pw_text(thePixwin,10, 10,
                PIX_SRC | PIX_COLOR(BLUE), NULL, "Multilayer Neural Net");
        DISPLAY_NET(thePixwin);
        break;
     case FINAL:
           hold_one_out();
           file_saliency(0);
           do_avg();
           write_calvin_weights("weights.calvin");
           break;
   }

}
```

6. *Initial.c* and *Control.c*

Initial.c contains the global data declarations needed by the paradigm class common routines. *Control.c* consists of functions to control the execution of the simulation: the control panel, IPC with ANNS, and IPC with the main

algorithm background procedure. Client-programmers only need to modify the name of the included header files in these two files.

B.4.2 **BackProp** *Subdirectory.* This directory is an example of NN paradigm component. Client-programmers may take this as an example for adding new NN paradigm to ANNS. Figure 58 shows the module diagram for the BackProp paradigm.

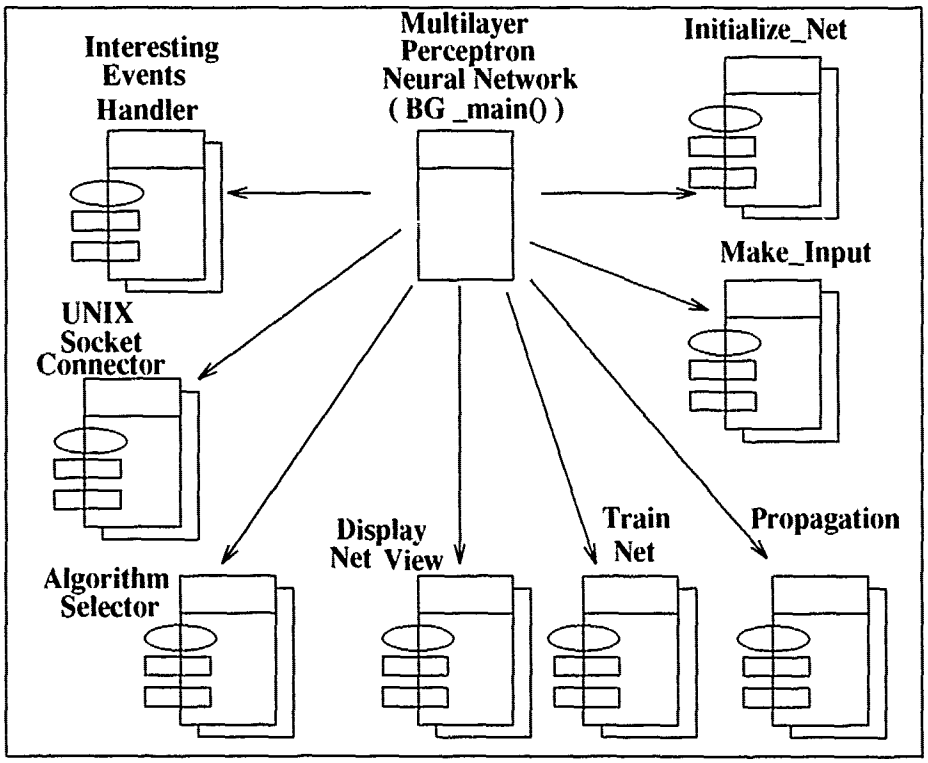


Figure 58. Module Diagram for BackProp NN subsystem

In pseudo-code the general flow of the BackProp paradigm is shown below:

```
begin
INITIALIZE
loop {
     MAKE_INPUT
     PROPAGATE
     TEST
     TRAIN_NET
```

```
        DISPLAY_NET
        Event Handler}
    end loop
    end
```

Appendix C. ANNS User Evaluation Form

CAD-Tool Human-Computer Interface Evaluation[1]

Name (administrative use only): _____.

Estimated time spent with tool/system [:]

Do not write in these spaces
Tool Evaluated:
Class:
Group:
Exper:
First:
ID#:

PLEASE READ BEFORE PROCEEDING:

The following questionnaire is designed to provide user feedback on the human-computer interface of the specified computer-aided design (CAD) tool. Through your responses, we hope to measure your degree of satisfaction with the tool, with primary emphasis on the "user-friendliness" of the human-computer interface.

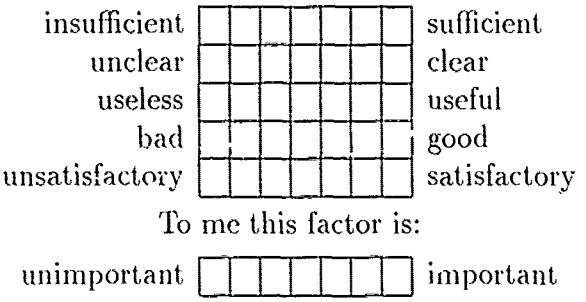
The questionnaire consists of a set of 11 factors, plus an overall rating. We will determine your satisfaction with the tool based on your response to six adjective pairs used to describe each factor. Each adjective pair has a seven-interval range where you are to indicate your feelings with an "X". Responses placed in the center of the range will

[1] U S Air Force Institute of Technology, AFIT/ENG

indicate that you have no strong feelings one way or the other, *or that you cannot effectively evaluate that given factor.*

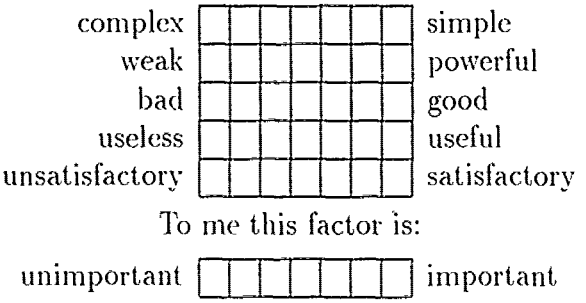
Evaluation begin time [:_____]

1. *System Feedback or Content of the Information Displayed.* The extent to which the system kept you informed about what was going on in the program.



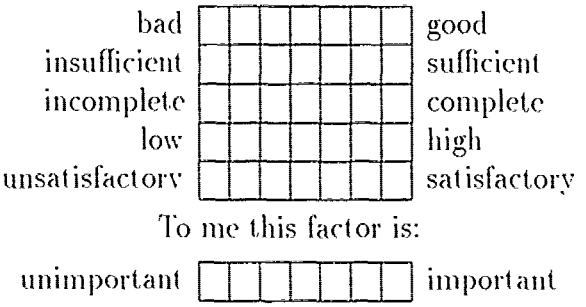
Comments:

2. *Communication.* The methods used to communicate with the tool.



Comments:

3. *Error Prevention* Your perception of how well the system prevented user induced errors.



Comments:

4. *Error Recovery.* The extent and ease with which the system allowed you to recover from user induced errors.

unforgiving	☐☐☐☐☐☐☐	forgiving
incomplete	☐☐☐☐☐☐☐	complete
complex	☐☐☐☐☐☐☐	simple
slow	☐☐☐☐☐☐☐	fast
unsatisfactory	☐☐☐☐☐☐☐	satisfactory

To me this factor is:

unimportant	☐☐☐☐☐☐☐	important

Comments:

5. *Documentation.* Your overall perception as to the usefulness of documentation.

useless	☐☐☐☐☐☐☐	useful
incomplete	☐☐☐☐☐☐☐	complete
hazy	☐☐☐☐☐☐☐	clear
insufficient	☐☐☐☐☐☐☐	sufficient
unsatisfactory	☐☐☐☐☐☐☐	satisfactory

To me this factor is:

unimportant	☐☐☐☐☐☐☐	important

Comments:

6. *Expectations.* Your perception as to the services provided by the system based on your expectations.

displeased	☐☐☐☐☐☐☐	pleased
low	☐☐☐☐☐☐☐	high
uncertain	☐☐☐☐☐☐☐	definite
pessimistic	☐☐☐☐☐☐☐	optimistic
unsatisfactory	☐☐☐☐☐☐☐	satisfactory

To me this factor is:

unimportant ☐☐☐☐☐☐☐ important

Comments:

7. *Confidence in the System.* Your feelings of assurance or certainty about the services provided by the system.

low	☐☐☐☐☐☐☐	high
weak	☐☐☐☐☐☐☐	strong
uncertain	☐☐☐☐☐☐☐	definite
bad	☐☐☐☐☐☐☐	good
unsatisfactory	☐☐☐☐☐☐☐	satisfactory

To me this factor is:

unimportant ☐☐☐☐☐☐☐ important

Comments:

8. *Ease of Learning.* Ease with which you were able to learn how to use the system to perform the intended task.

difficult	☐☐☐☐☐☐☐	easy
confusing	☐☐☐☐☐☐☐	clear
complex	☐☐☐☐☐☐☐	simple
slow	☐☐☐☐☐☐☐	fast
unsatisfactory	☐☐☐☐☐☐☐	satisfactory

To me this factor is:

unimportant ☐☐☐☐☐☐☐ important

Comments:

9. *Display of Information.* The manner in which both program control and data information were displayed on the screen.

confusing		clear
cluttered		well defined
incomplete		complete
complex		simple
unsatisfactory		satisfactory

To me this factor is:

unimportant		important

Comments:

10. *Feeling of Control.* Your ability to direct or control the activities performed by the tool.

low		high
insufficient		sufficient
vague		precise
weak		strong
unsatisfactory		satisfactory

To me this factor is:

unimportant		important

Comments:

11. *Relevancy or System Usefulness.* Your perception of how useful the system is as an aid to a software developer.

useless		useful
inadequate		adequate
hazy		clear
insufficient		sufficient
unsatisfactory		satisfactory

To me this factor is:

unimportant		important

Comments:

12. *Overall Evaluation of the System.* Your overall satisfaction with the system.

unsatisfied ☐☐☐☐☐☐☐ satisfied

(cont'd)

Comments on the Overall System:

Evaluation end time [:]

Total time spent on evaluation [:]

Thank you for your help.

Appendix D. The ANNS Source Codes

Bibliography

1. Boehm, Barry W. "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, 61–72 (May 1988).
2. Booch, Grady. "Object-Oriented Development," *IEEE Transaction on Software Engineering*, *SE-12*:211–221 (February 1986).
3. Booch, Grady. *Software Components with Ada*. Menlo Park CA: The Benjamin/Cummings Publishing Company, Inc. 1987.
4. Booch, Grady. *Software Engineering with Ada* (Second Edition). Menlo Park CA: The Benjamin/Cummings Publishing Company, Inc. 1987.
5. Booch, Grady. *Object Oriented Design with Applications*. Redwood City CA: The Benjamin/Cummings Publishing Company, Inc. 1991.
6. EVB Software Engineering, Inc. *An Object Oriented Design Handbook*. EVB Software Engineering, Inc. 1985.
7. Garth, S. "A Dedicated Computer for Simulation of Large Systems of Neural Nets," *Neural Computers*. 435–444 (Springer 1989).
8. Gottfried, Byron S. *Programming with C*. New York, New York: McGraw-Hill Publishing Company. 1990.
9. Heller, Dan. *XView Programming Manual*. Sebastopol CA: O'Reilly & Associates, Inc. 1991.
10. Henderson-Sellers, Brain and Julian M. Edwards. "The Object-Oriented Systems Life Cycle," *Communications of the ACM*, *33*:142 – 159 (September 1990).
11. Huang, W. Y. and R. P. Lippmann. "Neural Net and Traditional Classifiers," *Proceeding of the Conference on Neural Information Processing Systems* (November 1987).
12. Humphrey, Watts S. *Managing the Software Process*. Massachusetts: Addison-Wesley Publishing Company, Inc.. 1989.
13. Jean, Catherine and Alfred Strohmeier. "An experience in teaching OOD for ADA software," *Software Engineering Notes*, *15*:44 – 99 (October 1990).
11. Johnson, Eric F. and Kevin Reichard. *X Window Applications Programming*. Portland: MIS Press, 1989.
15. Jones, Oliver. *Introduction to the X Window System*. Englewood Cliffs NJ: Prentice Hall. 1989.
16. Kernighan, Brian W. and Dennis W. Richie. *The C Programming Language*. MA: Prentice Hall, Inc. 1988.
17. Koivunen, Marja-Ritta and Martii Mantyla. "HutWindows: An Improved Architecture for a User Interface Management System," *IEEE Computer Graphics and Applications*. 43 – 52 (January 1988).

18. Korson, Time and John D. McGregor. "Understanding Object Oriented: A Unifying Paradigm," *Communications of the ACM*, *33*:40 – 60 (September 1990).

19. Lippmann, Richard P. "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine* (April 1987).

20. Lowgren, Jonas. "History, State and Future of User Interface Management Systems," *SIGCHI Bulletin*, *20*:32 - 44 (July 1988).

21. Mackie, S., H.P. Graf and Schwartz D. B. "Implementations of Neural Network Models in Silicon," *Neural Computers*, 467–476 (Springer 1989).

22. Myers, Brad A. "A Taxonomy of Window Manager User Interfaces," *IEEE Computer Graphics and Applications*, *8*:65–84 (September 1988).

23. Myers, Brad A. *Software Design: User Interface Design (1), Video tape number AC-SD-01-24*. Carnegie Mellon University, Software Engineering Institute, 1989.

24. Myers, Brad A. *Software Design: User Interface Design (2), Video tape number AC-SD-01-25*. Carnegie Mellon University, Software Engineering Institute, 1989.

25. Myers, Brad A. and Mary Beth Rosson. "User Interface Programming Survey," *SIGCHI Bulletin*, *23*:27 - 30 (April 1991).

26. Nye, Adrian. *Xlib Programming Manual*. Sebastopol CA: O'Reilly & Associates, Inc, 1991.

27. Open Software Foundation, Englewood Cliffs, New Jersey. *OSF/Motif™ Programmer's Guide*, 1990.

28. Pountain, Dick. "The X Window System," *Byte*, *14*:353–360 (January 1989).

29. Pressman, Rogers S. *Software Engineering, A Practitioners Approach*. New York, New York: McGraw-Hill, Inc, 1987.

30. Raalte, Thomas Van, editor. *XView Reference Manual*. Sebastopol CA: O'Reilly & Associates, Inc. 1991.

31. Roberts, Stephen D. and Joe Heim. "A perspective on object-oriented simulation," *Proceedings of the 1998 Winter Simulation Conference*, 277 - 281 (1988).

32. Rogers, Steven K. and Matthew Kabrisky. *An Introduction to Biological and Artificial Neural Networks for Pattern Recognition*. Wright Patterson AFB, OHIO: Air Force Institute of Technology, 1989.

33. Rubin, Robert v., James Walker II and Eric Golin. "Design and Implementation of Programming Environments in the Visual Programmers Workbench," *Proceedings of the 14th Annual International Computer Software and Applications Conference*, 547–554. Piscataway, NJ: IEEE Press, 1990.

34. Ruck, Dennis W., et al. "The Multilayer Perceptron: A Bayes Optimal Discriminant Function Approximator.," *IEEE Transations on Neural Networks* (1, March 1990).

35. Rumbaugh, James and others. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

36. Scheiflier, Robert W. and others. *The X Window System: C Library and Protocol Reference*. Digital Press, 1988.

37. Scheiflier, Robert W. and Jim Gettys. "The X Window System," *ACM Transactions on Graphics*, *5*:79–109 (April 1986).

38. Seidewitz, Ed and Mike Stark. "Towards a General Object-Oriented Software Development Methodology," *Ada Letters*, *VII*:54 – 67 (July, August 1987).

39. Silicon Graphics, Inc. *Graphics Library Programming Guide*, 1989.

40. Silicon Graphics, Inc. *Graphics Library Reference Manual*, 1989.

41. Snith, Sydney L. and Jane N. Mosier, *Guidelines for Designing User Interface Software*. Bedford MA: MITRE Corporation, August 1986. Contract F19628-86-C-0001.

42. Sommerville, Ian. *Software Engineering* (Third Edition). Massachusetts: Addison-Wesley Publishing Company, 1989.

43. Sun Microsystems, Inc. *Network Programming*, 1990.

44. Sun Microsystems, Inc. *OpenWindows Version 2 Release Notes*, 1990.

45. Sun Microsystems, Inc. *Programming Utilities and Libraries*, 1990.

46. Sun Microsystems, Inc. *SunOS Reference Manual*, 1990.

47. Sun Microsystems, Inc. *SunView Programmer's Guide*, 1990.

48. Sun Microsystems, Inc. *SunView System Programmer's Guide*, 1990.

49. SunSoft, Div of Sun Microsystems. *OpenWindows™ Version 3 for SunOS™ 4.1.x*, 1991.

50. Szekely, Pedro. "Separating the User Interface from the Functionality of Application Programs," *SIGCHI Bulletin*, *18*:45 – 46 (October 1986).

51. Tarr, Gregory L. *Dynamic Analysis of Feedforward Neural Networks using Simulated and Measured Data*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988.

52. Tarr, Gregory L. "AFIT Neural Network Development Tools and Modeling Artificial Neural Networks," *SPIE Symposium on Applications of Artificial Neural Networks* (1989).

53. Tarr, Gregory L., et al. "Effective Neural Network Modeling in C," *Proceedings of the 1991 International Conference on Artificial Neural Networks* (June 1991).

54. Trumbly, James E. and Kirk P. Arnett. "Including a User Interface Management Systems (UIMS) in the Performance Relationship Model," *SIGCHI Bulletin*, *20*:56-62 (April 1989).

55. Unger, Brain W. "Object oriented simulation-Ada, C++, Simula," *Proceedings of the 1986 Winter Simulation Conference*, 123 - 124 (1986).

56. Wu, Ching-seh. "ANNS Programmer's Guide," *In Publication* (June 1993).

57. Wu, Ching-seh. "ANNS User's Manual," *In Publication* (June 1993).

58. Wu, Ching-seh, et al. "A Public Domain X Window Based Artificial Neural Network Simulator," *Submitted for Publication* (June 1993).

59. Young, Douglas. *X Window Systems: Programming and Applications with Xt*. Englewood Cliffs NJ: Prentice Hall, 1989.

60. Yourdon, Edward. *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall, Inc, 1989.

Vita

Captain Ching-seh Wu was born on 24 January 1963 in Kaohsiung, Taiwan, Republic of China (R.O.C.). He graduated from Tzuoying High School in Kaohsiung, a southern city of Taiwan, in June 1981. He received a Bachelor of Science in Surveying Engineering degree at the Chun-Chen Institute of Technology in Taoyuan, a northern city of Taiwan, in June 1986. After graduation, he was assigned as a first lieutenant of the R.O.C. Air Force and worked as an Intelligence officer at Taoyuan Air Force Base. In March 1989, he came to United States and attended an Intelligence Officer Training course for three months at Goodfellow Air Force Base in Texas. He was accepted into the R.O.C. Air Force Education Program and entered the School of Engineering at Air Force Institute of Technology (AFIT) of United States in June, 1991. One month before coming to United States, Captain Wu got married to Pi-chiao (Joy) Yu and now has a Chinese-American baby: Kevin Wu.

Upon completion of his graduate studies at the Air Force Institute of Technology, Captain Wu will begin an assignment at the R.O.C. Air Force Academy.

Permanent address: 16 SanMing Rd. KungSang
Kaohsiung, Taiwan, R.O.C.

June 1993 Master's Thesis

ANNS-An X Window Based Version of the AFIT Neural Network Simulator

Ching-seh Wu

Air Force Institute of Technology, WPAFB OH 45433-6583

AFIT/GCE/ENG/93J-01

Abstract

This thesis presents an X Window based neural network simulation environment developed at Air Force Institute of Technology (AFIT) using the techniques of modern software engineering . This artificial neural network simulator is a tool running on Sun SPARCstations and supporting two user modes: end-users and client-programmers. End-users interact with neural network paradigms developed by client-programmers for the purpose of studying and analyzing the execution of a particular Neural Network (NN) paradigm, or class of NN algorithms. Client programmers maintain the system and use this environment for the development of new NN paradigms or algorithms for end-users. The development follows a hybrid software engineering paradigm which combines the best characteristics of the classic life cycle, prototype, and iterative methodologies through *requirements, design, implementation, and testing.* An object-oriented approach is used for the design including preliminary and detailed design. The system is implemented with the C programming language on Sun workstation and uses the XView window-based environment. It provides users with a variety of control and input options: simulation speed control, multiple and simultaneous NN algorithm simulations, and simulation environment control.

Neural Network Simulator, Xview, C 172

UNCLASSIFIED UNCLASSIFIED UNCLASSIFIED UL